Trading Bot System Design Principles

A Comprehensive Research Report

Page 1: Overview & Classification

Executive Summary

This report documents the 14 foundational system design principles applied in building a production-grade algorithmic trading bot. These principles span operating systems theory, distributed systems, concurrency control, and software engineering best practices. The architecture implements a monolithic, in-memory, event + time coordinated system with explicit state ownership—the canonical starting point for professional trading systems before horizontal scaling becomes necessary.

Classification of Principles

The 14 principles are organized into three supersets:

Superset A: Core Architecture Patterns (Principles 1-6)

High-level structural decisions about how the system is organized and how components interact.

1. Separation of Concerns
2. Event-Driven Architecture
3. Caching Pattern
4. Producer-Consumer Pattern
5. Stateful Service
6. Fail-Fast Design

Superset B: Advanced OS & Concurrency Principles (Principles 7-12)

Low-level principles derived from operating systems, concurrent programming, and distributed systems theory.

7. Process Isolation
8. Single Source of Truth (SSOT)
9. Single-Writer, Multiple-Reader (SWMR)
10. Shared-Nothing Across Processes, Shared-Everything Inside Process
11. Replace, Don't Mutate (Atomic State Transitions)
12. Temporal Determinism

Superset C: Engineering Discipline & Pragmatism (Principles 13-14)

Practical engineering wisdom about when to add complexity and how to maintain system observability.

13. YAGNI (You Aren't Gonna Need It)
14. Fail-Fast, Debuggable Systems

System Architecture Context

Domain: Algorithmic equity trading on Indian markets (NSE/BSE)
Scale: 5 stocks, 5-minute candles, <500ms processing latency
Components: APScheduler (data pipeline), WebSocket (tick stream), ML inference, order execution
Technology Stack: Python, XGBoost, SQLite, Fyers API
Deployment: AWS EC2 (single instance, Ubuntu)

Page 2: Separation of Concerns

First Principles

In any complex system, cognitive load increases exponentially with the number of interacting responsibilities. By isolating distinct concerns into separate modules, we reduce coupling, improve testability, and enable independent evolution of components.

Definition

Separation of Concerns (SoC) is a design principle stating that a system should be divided into distinct sections, each addressing a separate concern. A concern is a specific aspect of functionality—what the code does, not how it's organized physically.

Layman's Explanation

Imagine building a house. You don't have one person doing plumbing, electrical, framing, and roofing simultaneously. Each tradesperson focuses on their specialty. When the electrician needs to change wiring, they don't need to understand plumbing. Similarly, in software, the module that fetches data shouldn't need to know how trades are executed.

Example 1: MVC Pattern in Web Applications

In a Django web app:

- **Model** (concerns: data structure, database queries)
- **View** (concerns: business logic, request handling)
- **Template** (concerns: HTML rendering, user interface)

A database schema change doesn't require modifying templates. UI redesigns don't touch database code.

Example 2: Unix Philosophy

Unix commands follow SoC religiously:

- `grep` searches text
- `sort` orders lines
- `uniq` removes duplicates

You combine them: `cat log.txt | grep ERROR | sort | uniq`

Each tool does one thing well. Changes to sorting algorithms don't affect grep.

Application in Our Trading Bot

Our system separates three core concerns:

1. Data Pipeline (scheduler.py)

- Concern: Fetch OHLCV data, compute features, run ML inference
- Runs every 5 minutes
- Updates SQLite + RAM state
- Independent of execution logic

2. Signal Detection (websocket_handler.py)

- Concern: Monitor tick data, detect entry conditions
- Runs continuously during market hours
- Reads from RAM state
- Independent of how features are computed

3. Trade Execution (execution.py)

- Concern: Place orders, manage positions, calculate stop-loss/target
- Triggered by signal detection
- Independent of data pipeline implementation

Benefits Realized:

- Can replace SQLite with TimescaleDB without touching WebSocket code
- Can switch brokers (Fyers → Zerodha) without modifying ML pipeline
- Can test signal detection logic without running actual scheduler

- Can upgrade XGBoost models without changing execution module

Anti-pattern Avoided: Putting feature calculation inside WebSocket handler would create tight coupling—every tick would trigger expensive computations, and changing feature logic would risk breaking real-time signal detection.

Page 3: Event-Driven Architecture

First Principles

Systems can be organized around two paradigms: request-response (pull) or event-driven (push). In domains where state changes originate externally and unpredictably, event-driven architectures reduce latency and resource waste by reacting to changes rather than polling for them.

Definition

Event-Driven Architecture (EDA) is a software design pattern in which system components react to events (significant state changes) rather than continuously polling for changes. Events are notifications that something has occurred; listeners subscribe to events and execute logic when triggered.

Layman's Explanation

Think of a doorbell vs. repeatedly checking if someone is at the door. With a doorbell (event-driven), you go about your activities and respond when the bell rings. Without it (polling), you'd have to check the door every 10 seconds—wasteful and delayed.

Similarly, instead of checking every millisecond "did the price change?", the market pushes notifications: "Price changed to ₹1,234.50" → your system reacts.

Example 1: GUI Event Handling

In a desktop application:

```python
button.on_click(lambda: save_file())
```

The program doesn't check every nanosecond "was button clicked?". The OS fires a click event → callback executes. This is how all modern UIs work—event loops waiting for mouse/keyboard events.

Example 2: AWS Lambda + S3

```
User uploads image → S3 fires event → Lambda resizes image → stores thumbnail
```

Lambda doesn't poll S3 asking "new file?". S3 emits event, Lambda reacts. Serverless architectures are fundamentally event-driven.

Application in Our Trading Bot

Our system implements EDA at multiple levels:

Level 1: WebSocket Tick Events

```python
def on_tick(ticker, ltp):
    # Event: New LTP received
    if ltp <= MARKET_STATE[ticker]['support']:
        check_conditions(ticker, ltp)
```

Market sends tick → WebSocket fires event → condition check runs. No polling required.

Level 2: APScheduler Time Events

```python
@scheduler.scheduled_job('interval', minutes=5)
def pipeline():
    # Event: 5 minutes elapsed
    fetch_and_compute_features()
```

Scheduler fires time-based event every 5 minutes → data pipeline runs. Not polling clock.

Level 3: Condition Met Events

```python
def check_conditions(ticker, ltp):
    if all_conditions_met:
        # Event: Entry conditions satisfied
        execute_trade(ticker)
```

Conditions met → execution triggered. Not continuously checking "can I trade yet?".

Benefits Realized:

- **Latency:** React to price changes in <10ms (WebSocket push) vs. 100ms+ (polling)
- **Resource Efficiency:** CPU idle when no events, not burning cycles polling
- **Scalability:** Can handle 50 tickers with same architecture—events don't compound like polling loops
- **Determinism:** Each event has clear causality chain (tick → condition check → trade)

Trade-off: Event-driven systems are harder to debug (non-linear execution flow) vs. sequential polling. We address this with comprehensive logging of event chains.

Page 4: Caching Pattern

First Principles

Data access exhibits temporal locality—recently accessed data is likely to be accessed again soon. Spatial locality also applies—data near recently accessed data is likely needed. By maintaining a fast-access copy of frequently used data, we minimize expensive retrieval operations.

Definition

Caching Pattern is a performance optimization technique where a subset of data is stored in a faster storage medium (cache) while maintaining a complete copy in slower persistent storage (backing store). Reads check cache first; writes update both. This is a classic time-space tradeoff.

Layman's Explanation

Imagine you're cooking and need salt. You could walk to the grocery store each time (slow, persistent storage). Instead, you keep salt in your kitchen cabinet (fast cache). When you run out, you restock from the store. Your cabinet doesn't hold everything, just what you use frequently.

Similarly, our database has all historical data (grocery store), but RAM holds just the latest values (kitchen cabinet) for instant access.

Example 1: CPU Cache Hierarchy

Modern processors have L1/L2/L3 caches:

- **L1 cache:** 64KB, 1ns access
- **RAM:** 16GB, 100ns access
- **SSD:** 1TB, 100,000ns access

Frequently accessed variables stay in L1. When you loop over an array, the CPU pre-fetches nearby elements into cache (spatial locality).

Example 2: CDN (Content Delivery Network)

Netflix stores movies on origin servers (backing store). When you watch Stranger Things, your ISP's CDN caches it (fast local copy). Second viewer in your city streams from CDN, not origin—reducing latency from 500ms to 20ms.

Application in Our Trading Bot

We implement a **write-through cache** pattern:

Backing Store: SQLite Database

- Holds complete historical data (300 candles × 5 tickers)
- All features for all timestamps
- ML predictions for entire dataset
- ~50MB per day
- Access time: 5-10ms (disk I/O)

Cache: RAM (MARKET_STATE dict)

- Holds only latest row per ticker
- 5 tickers × 50 features × 8 bytes = ~2KB total
- Access time: <1µs (memory lookup)
- Updated every 5 minutes

Write-Through Logic:

```python
def pipeline_function(ticker):
    final_df = create_features(df)
    prediction = get_latest_prediction(ticker, final_df)

    # Write to backing store (SQLite)
    save_features_df(ticker, final_df)

    # Write to cache (RAM)
    MARKET_STATE[ticker] = extract_latest(final_df)
```

Read Pattern:

```python
# WebSocket reads from cache (not database)
def on_tick(ticker, ltp):
    support = MARKET_STATE[ticker]['support']  # RAM lookup: <1µs
    # Not: support = query_sqlite(ticker, 'support')  # Disk I/O: 5ms
```

Cache Invalidation: Timer-based every 5 minutes—no stale data since market data has natural time boundaries.

Benefits Realized:

- **Speed:** 5000x faster reads (1µs vs 5ms)
- **Throughput:** Can handle 10,000 ticks/second without database bottleneck
- **Reliability:** Database can be down temporarily; trading continues from cache
- **Historical Analysis:** Full dataset in SQLite for backtesting, not just latest snapshot

Why Not Cache Everything?

- Only need latest values for live trading
- Full dataset (300 candles) would be 2KB × 300 = 600KB—still tiny, but unnecessary
- Principle: cache only working set, not entire dataset

Page 5: Producer-Consumer Pattern

First Principles

In concurrent systems, when one component generates data faster or slower than another can process it, directly coupling them creates blocking and resource waste. By introducing a buffer between producer and consumer, we decouple their execution rates and enable asynchronous processing.

Definition

Producer-Consumer Pattern is a synchronization pattern where one or more producer threads generate data/events and place them in a shared buffer, while one or more consumer threads retrieve and process that data. The buffer (queue) decouples production rate from consumption rate.

Layman's Explanation

Imagine a bakery. Bakers (producers) make bread continuously and place it on a shelf (buffer). Cashiers (consumers) sell bread to customers at varying rates. If bakers waited for each loaf to be sold before baking the next, they'd be idle. The shelf decouples baking speed from selling speed.

Similarly, our scheduler produces market data every 5 minutes. WebSocket consumes that data at tick speed (milliseconds). They operate independently via shared state (the buffer).

Example 1: Print Spooler

When you print a document:

- **Producer:** Your application sends print jobs to spooler

- **Buffer:** Print queue on disk
- **Consumer:** Printer driver processes queue

You can queue 10 documents instantly (producer fast), while printer prints 1 page/second (consumer slow). No blocking.

Example 2: Apache Kafka

```
Producers (web servers) → Kafka topics (buffer) → Consumers (analytics pipeline)
```

Web servers log events at 10,000/sec. Analytics processes 100/sec. Kafka buffers millions of events, allowing producers and consumers to operate at different rates.

Application in Our Trading Bot

Our implementation uses **state as the buffer** rather than a traditional queue:

Producer: APScheduler

- Generates features + ML predictions every 5 minutes
- Production rate: 1 update / 300 seconds
- Writes to MARKET_STATE (buffer)

Consumer: WebSocket Handler

- Reads features to check conditions
- Consumption rate: 100-1000 reads / second (tick frequency)
- Reads from MARKET_STATE (buffer)

Asymmetric Rates:

- Producer: 0.0033 updates/sec (1 every 5 min)
- Consumer: 500 reads/sec (during active trading)
- **Ratio:** Consumer reads 150,000x more frequently than producer writes

Traditional Queue vs. State Buffer:

If we used a queue:

```python
```

```
# Producer
queue.put({'ticker': 'RELIANCE', 'support': 1234, ...})

# Consumer
while True:
    data = queue.get()  # Blocks until new data
    check_conditions(data)
```

Problem: Consumer blocks when queue empty. With 5-minute update intervals, WebSocket would be blocked 99.9% of the time.

Our state-based approach:

```python
# Producer (non-blocking write)
MARKET_STATE['RELIANCE'] = latest_features

# Consumer (non-blocking read, uses cached value)
support = MARKET_STATE['RELIANCE']['support']
```

Consumer always reads latest state without blocking. No queue needed because we don't need every historical update—only the current state matters for trading decisions.

Benefits Realized:

- **Decoupling:** Scheduler can be stopped/restarted without affecting WebSocket
- **Rate Independence:** Producer and consumer operate at natural speeds
- **No Blocking:** WebSocket never waits for scheduler
- **Simplicity:** State buffer simpler than queue for latest-value semantics

Why This Works for Trading: We care about "what is the current support level?"—not "give me all historical support levels in order". State buffer is semantically correct for our domain.

Page 6: Stateful Service

First Principles

Services can be stateless (pure functions, no memory between requests) or stateful (maintain context across interactions). Stateless scales horizontally easily but requires external state storage. Stateful services trade scalability for performance by keeping working set in memory.

Definition

Stateful Service is an architectural pattern where the application maintains internal state (in RAM) across multiple operations. The service "remembers" previous interactions and uses that context to make decisions. This contrasts with stateless services that treat each request independently.

Layman's Explanation

Think of two restaurant experiences:

Stateless (Fast Food): Each order independent. You say "burger" every time, cashier doesn't remember you.

Stateful (Personal Chef): Chef remembers you like burgers medium-rare, no pickles. Uses that memory to customize without you repeating preferences.

Our trading bot is the personal chef—it remembers current support levels, RSI, open positions, so each tick doesn't require fetching all context from database.

Example 1: Video Game

A game maintains state:

- Player position, health, inventory
- Enemy locations
- Game world changes

If the game were stateless, every frame would reload the entire game state from disk—impossible at 60 FPS. Instead, state lives in RAM, persisted to disk only at save points.

Example 2: HTTP Session vs. JWT

Stateful (Session):

```
Server stores: sessions[token] = {user_id: 123, cart: [...]}
Client sends: Cookie: session=abc123
Server reads: sessions['abc123'] → knows user state
```

Stateless (JWT):

```
Client sends: Bearer eyJ0eXAiOiJKV1QiLCJh... (entire state encoded)
Server decodes JWT each request, no memory of user
```

Stateful is faster (RAM lookup) but doesn't scale horizontally. Stateless scales but adds latency (decode JWT each time).

Application in Our Trading Bot

We deliberately chose **stateful architecture** for performance:

State Maintained in RAM:

```python
MARKET_STATE = {
  'RELIANCE': {
    'support': 1234.50,
    'resistance': 1250.00,
    'rsi': 45.2,
    'ma_50': 1235.0,
    'ml_prediction': 1,
    'ml_confidence': 0.75
  },
  # ... 4 more tickers
}
```

Per-Request Cost Comparison:

Stateful (our approach):

```python
# On each tick (1000x per second)
support = MARKET_STATE['RELIANCE']['support']  # 1μs RAM lookup
```

Stateless (alternative):

```python
# On each tick
support = query_sqlite("SELECT support FROM features WHERE ticker='RELIANCE' ORDER BY timestamp DESC LIMI
# 5ms disk I/O × 1000 ticks/sec = 5 seconds of DB time per second → impossible
```

Why Stateful Works for Us:

1. **Single Instance:** Running on one EC2 server, no horizontal scaling needed
2. **Small State:** 5 tickers × 50 features = 2KB, fits in L1 cache
3. **High Read Frequency:** 1000 reads/sec × 5ms DB latency = unacceptable
4. **Deterministic Updates:** State updates every 5 minutes, predictable refresh

Trade-offs Accepted:

- **No Horizontal Scaling:** Can't easily run 10 instances behind load balancer
- **Restart Cost:** On restart, must reload state (5-second warmup)
- **Single Point of Failure:** If process crashes, state lost (acceptable—rebuilds in 5 min)

When We'd Go Stateless: If scaling to 500 tickers across 10 servers, we'd use:

- Redis for shared state
- Stateless workers reading from Redis
- Horizontal scaling behind load balancer

But for 5 tickers on 1 server, stateful is 5000x faster.

Page 7: Fail-Fast Design

First Principles

In systems with sequential dependencies, early termination upon failure reduces wasted computation and improves debuggability. By checking preconditions first and exiting immediately on failure, we avoid executing expensive operations that will ultimately fail anyway.

Definition

Fail-Fast Design is a system design principle where failures are detected as early as possible and execution halts immediately rather than continuing with invalid state. This prevents cascading failures, reduces resource waste, and makes error conditions explicit.

Layman's Explanation

Imagine cooking a recipe:

Non-Fail-Fast: Start cooking, chop vegetables, heat pan, then discover you're out of salt → wasted 20 minutes.

Fail-Fast: Check all ingredients first. If missing salt, stop immediately → save 20 minutes.

Similarly, our trading bot checks conditions in order of cost—cheapest first (LTP comparison), most expensive last (trade execution).

Example 1: Input Validation

Bad (not fail-fast):

```python
```

```
def process_user(data):
    user = create_user(data)  # Expensive DB write
    send_email(user.email)     # Expensive API call
    if not valid_email(user.email):  # Should check first!
        raise ValueError("Invalid email")
```

Good (fail-fast):

```python
def process_user(data):
    if not valid_email(data['email']):  # Check FIRST
        raise ValueError("Invalid email")
    user = create_user(data)
    send_email(user.email)
```

Example 2: Database Connection

Bad:

```python
query = prepare_complex_query()  # 1 second
results = db.execute(query)      # Fails: connection closed
```

Good:

```python
if not db.is_connected():        # Fail immediately
    raise ConnectionError()
query = prepare_complex_query()
results = db.execute(query)
```

Application in Our Trading Bot

We implement fail-fast at three levels:

Level 1: Tick-Level (cheapest check)

```python
```

```python
def on_tick(ticker, ltp):
    # Check 1: Price at support? (1µs - RAM comparison)
    if ltp > MARKET_STATE[ticker]['support']:
        return  # Fail fast - 99% of ticks exit here

    # Check 2: RSI in range? (1µs - RAM lookup)
    rsi = MARKET_STATE[ticker]['rsi']
    if not (35 < rsi < 55):
        return  # Fail fast - 90% of remaining ticks exit

    # Check 3: Trend bullish? (1µs - RAM lookup)
    if MARKET_STATE[ticker]['ma_50'] <= MARKET_STATE[ticker]['ma_200']:
        return  # Fail fast

    # All conditions met - proceed to expensive operations
    check_ml_filter(ticker)
```

Cost Savings:

- 99% of ticks fail first check: 1µs wasted (vs. 10ms if we queried DB)
- 0.9% pass to second check: still just RAM lookups
- 0.01% pass all checks: only then do we proceed to execution

Level 2: ML Filter (moderate cost)

```python
def check_ml_filter(ticker):
    # Already in RAM from 5-min update
    if MARKET_STATE[ticker]['ml_prediction'] != 1:
        log_rejected_signal(ticker)
        return  # Fail fast - don't execute

    execute_trade(ticker)
```

Level 3: Execution (expensive - only if all checks pass)

```python
```

```python
def execute_trade(ticker):
    # Final sanity checks before money leaves
    if position_already_open(ticker):
        return  # Fail fast

    if insufficient_capital():
        return  # Fail fast

    place_order(ticker)  # Most expensive operation
```

Sequential Ordering by Cost:

1. LTP comparison: 1µs (RAM)
2. RSI/MA checks: 1µs each (RAM)
3. ML prediction: 1µs (RAM)
4. Position check: 100µs (dict lookup)
5. Capital check: 100µs (calculation)
6. Order placement: 50ms (network API call)

Benefits Realized:

- **Performance:** 99.99% of ticks handled in <5µs
- **Resource Efficiency:** Broker API called only on valid signals (1-10x per day)
- **Debugging:** Clear failure points—logs show exactly which condition failed
- **Safety:** Multiple sanity checks before money moves

Contrast with Non-Fail-Fast:

```python
```

```
# Anti-pattern: check everything, then decide
def on_tick(ticker, ltp):
    conditions = {
        'price': ltp <= support,
        'rsi': 35 < rsi < 55,
        'trend': ma50 > ma200,
        'ml': prediction == 1,
        'capital': check_capital(),
        'position': not has_position()
    }

    if all(conditions.values()):  # Evaluated ALL conditions first
        execute_trade()
```

This would call `check_capital()` and `has_position()` on every tick—wasteful.

Page 8: Process Isolation

First Principles

Modern operating systems provide memory isolation between processes to prevent one process from corrupting another's memory. This is a fundamental security and stability mechanism. Processes communicate via explicit IPC (Inter-Process Communication) mechanisms, not by accessing each other's memory.

Definition

Process Isolation is an OS-level principle where each process has its own virtual address space. Memory in one process is invisible and inaccessible to other processes unless explicitly shared via OS-provided mechanisms (shared memory, sockets, pipes, message queues).

Layman's Explanation

Think of apartments in a building. Each apartment (process) has its own space. You can't walk through your neighbor's wall to grab their TV (access their memory). If you want their TV, you knock on the door and ask (IPC).

Similarly, `python scheduler.py` and `python websocket.py` are separate apartments. They can't magically share variables just because they're in the same building (repository).

Example 1: Chrome Browser Tabs

Chrome runs each tab as a separate process:

- Tab 1 crashes → doesn't crash Tab 2

- Malicious JavaScript in Tab 1 can't read passwords from Tab 2
- OS enforces memory isolation

This is why Chrome uses 2GB RAM for 10 tabs—each process has overhead, but isolation is worth it.

Example 2: Docker Containers

```bash
docker run -d app1
docker run -d app2
```

`app1` and `app2` are isolated:

- Can't see each other's files
- Can't access each other's memory
- Can't interfere with each other's network ports

Must use Docker networks or volumes for communication.

Application in Our Trading Bot

Initial Misconception (corrected):

```
"If I have scheduler.py and websocket.py in the same folder,
they share variables automatically, right?"
```

Reality:

```python
# Running separately
$ python scheduler.py &    # Process ID: 1234
$ python websocket.py &    # Process ID: 5678

# Each gets own copy of state.py module
# Changes in PID 1234 invisible to PID 5678
```

Why This Matters:

```python
```

```
# state.py
MARKET_STATE = {}

# scheduler.py (Process 1)
from state import MARKET_STATE
MARKET_STATE['RELIANCE'] = {'support': 1234}

# websocket.py (Process 2)
from state import MARKET_STATE
print(MARKET_STATE)  # Output: {} (empty!)
```

Process 2's `MARKET_STATE` is a different memory location. Process 1's update invisible.

Our Solution: Single Process Architecture

```python
# main.py (ONE process)
import threading
from scheduler import run_scheduler
from websocket_handler import run_websocket
from state import MARKET_STATE  # Shared within process

if __name__ == "__main__":
    # Both threads in SAME process → SAME MARKET_STATE
    t1 = threading.Thread(target=run_scheduler)
    t2 = threading.Thread(target=run_websocket)

    t1.start()
    t2.start()
```

Respecting Process Isolation: We respect OS boundaries by:

1. **Not fighting isolation:** Don't expect cross-process sharing
2. **Choosing single-process:** Explicitly colocate to enable sharing
3. **Knowing the alternative:** If we needed multiple processes, we'd use Redis (explicit IPC)

When Process Isolation Helps Us: If WebSocket crashes due to network issue:

- In multi-process: only WebSocket dies, scheduler continues
- In single-process: entire bot crashes

We accept this trade-off because:

- Shared state is more important than isolation for our use case
- We use systemd to auto-restart on crash
- 5-minute recovery time acceptable (rebuilds state from SQLite)

Advanced Note: Python's `multiprocessing` module provides shared memory via:

```python
from multiprocessing import Manager
manager = Manager()
shared_dict = manager.dict()  # Explicit IPC, slower than threads
```

But this adds 10-100µs latency per access—unacceptable for tick processing.

Page 9: Single Source of Truth (SSOT)

First Principles

When multiple copies of the same data exist, they inevitably diverge. Determining which copy is "correct" becomes impossible. By designating one authoritative source for each piece of state, we eliminate inconsistencies and simplify reasoning about system behavior.

Definition

Single Source of Truth (SSOT) is a data management principle where each piece of information is stored in exactly one place, and all other references to that information point to or derive from that single source. Any reads or writes interact with the authoritative copy.

Layman's Explanation

Imagine a family calendar. If Dad has one calendar, Mom has another, and kids have their own, chaos ensues:

- Dad thinks soccer is Tuesday
- Mom thinks it's Wednesday
- Kid misses practice

Solution: One calendar on the fridge. Everyone consults and updates the same calendar. That's SSOT.

Similarly, our bot has one `MARKET_STATE` dict—not separate copies in scheduler, websocket, and execution modules.

Example 1: Git Repository

In a team project:

- **SSOT:** GitHub repository (origin/main branch)
- **Derived Copies:** Each developer's local clone

When you `git pull`, you sync with SSOT. When you `git push`, you update SSOT. If conflicts arise, Git forces resolution against SSOT—not arbitrary "which developer is right?"

Example 2: Accounting Ledger

In double-entry bookkeeping:

- **SSOT:** General ledger
- **Derived:** Account statements, reports, dashboards

All reports derive from the ledger. If a report shows different numbers than the ledger, the ledger is authoritative. You don't update reports directly; you correct ledger entries.

Application in Our Trading Bot

Our SSOT Structure:

```python
# state.py - THE authoritative source
MARKET_STATE = {
    'RELIANCE': {
        'support': 1234.50,      # ← SSOT for current support
        'rsi': 45.2,             # ← SSOT for current RSI
        'ml_prediction': 1       # ← SSOT for current prediction
    }
}
```

All Modules Reference This:

Scheduler (Writer):

```python
```

```python
from state import MARKET_STATE

def pipeline_function(ticker):
    features = compute_features()

    # Update SSOT
    MARKET_STATE[ticker] = features

    # NOT creating local copy:
    # scheduler_state[ticker] = features ❌
```

WebSocket (Reader):

```python
from state import MARKET_STATE

def on_tick(ticker, ltp):
    # Read from SSOT
    support = MARKET_STATE[ticker]['support']

    # NOT maintaining separate copy:
    # websocket_cache[ticker] = ... ❌
```

Execution (Reader):

```python
from state import MARKET_STATE

def execute_trade(ticker):
    # Read from SSOT
    atr = MARKET_STATE[ticker]['atr']
    stop_loss = entry - (atr * 2)

    # NOT storing ATR locally:
    # self.atr_values[ticker] = atr ❌
```

Anti-Pattern Avoided:

Bad (Multiple Sources of Truth):

```python
```

```
# scheduler.py
scheduler_features = {'RELIANCE': {'rsi': 45.2}}

# websocket.py
websocket_features = {'RELIANCE': {'rsi': 45.2}}  # Duplicate!

# Divergence over time:
# Scheduler updates → scheduler_features['RELIANCE']['rsi'] = 50.1
# WebSocket still has 45.2 → trades on stale data → losses
```

Consequences of Violating SSOT:

1. **Data Staleness:** WebSocket uses old RSI while scheduler has new value
2. **Race Conditions:** Which version is "correct" when they differ?
3. **Debugging Nightmare:** "Why did it trade? RSI was 60!" → "Which RSI? Scheduler had 55, WebSocket had 60."
4. **Synchronization Hell:** Now need locks, versioning, conflict resolution

How We Enforce SSOT:

1. **Import, Don't Copy:** All modules import same `MARKET_STATE`
2. **Single Writer:** Only scheduler writes features (SWMR pattern, next section)
3. **No Local Caches:** Modules read directly, don't maintain copies
4. **Immutable Reads:** Modules don't modify what they read (defensive programming)

Benefits Realized:

- **Consistency:** RSI value same across all modules at any instant
- **Debuggability:** One place to inspect state
- **Simplicity:** No synchronization logic needed
- **Correctness:** Impossible to trade on stale data (all reads see latest)

SQLite as Secondary SSOT:

- **RAM (MARKET_STATE):** SSOT for current state (last 5 minutes)
- **SQLite:** SSOT for historical state (all time)

These are separate domains—not duplicates. Current state lives in RAM, historical in DB.

Page 10: Single-Writer, Multiple-Reader (SWMR)

First Principles

Concurrent writes to shared state create race conditions, requiring locks that serialize access and degrade performance. If we can structure systems so only one writer updates state while many readers consume it, we eliminate write conflicts and enable lock-free reading.

Definition

Single-Writer, Multiple-Reader (SWMR) is a concurrency pattern where exactly one thread/process is allowed to write to a shared resource, while any number of threads can read concurrently. This eliminates write-write conflicts and often allows lock-free reads, dramatically improving performance.

Layman's Explanation

Think of a museum:

- **Guide (writer):** One person updates the exhibit descriptions overnight
- **Visitors (readers):** Thousands visit during the day, all reading simultaneously

Visitors don't interfere with each other (reading is safe). Guide works alone at night (no write conflicts). If we allowed multiple guides to update simultaneously, they'd overwrite each other's changes.

Similarly, only our scheduler updates `MARKET_STATE`. WebSocket and execution just read.

Example 1: DNS Server

In DNS infrastructure:

- **Writer:** Zone file updates (admin changes records)
- **Readers:** Millions of DNS queries per second

Writes are rare (hours between updates). Reads are massive. SWMR allows reads to happen without locking, achieving millions QPS.

Example 2: Linux Kernel RCU (Read-Copy-Update)

Linux uses RCU for lock-free reads:

```c
```

```
// Writer (rare)
new_data = copy_and_modify(old_data);
rcu_assign_pointer(global_ptr, new_data);
synchronize_rcu();  // Wait for readers
free(old_data);

// Readers (frequent)
rcu_read_lock();  // No actual lock, just marker
data = rcu_dereference(global_ptr);  // Lock-free read
use(data);
rcu_read_unlock();
```

Readers never block each other or writer. Writer ensures readers finish before freeing old data.

Application in Our Trading Bot

Our SWMR Architecture:

Single Writer: Scheduler

```python
# Only scheduler writes to MARKET_STATE
@scheduler.scheduled_job('interval', minutes=5)
def pipeline_function(ticker):
    features = compute_features()
    prediction = get_ml_prediction()

    # WRITE (only place this happens)
    MARKET_STATE[ticker] = {
        'support': features['support'],
        'rsi': features['rsi'],
        'ml_prediction': prediction
    }
```

Multiple Readers:

Reader 1: WebSocket

```python
```

```
def on_tick(ticker, ltp):
    # READ (no writes)
    support = MARKET_STATE[ticker]['support']
    rsi = MARKET_STATE[ticker]['rsi']
```

Reader 2: Execution

```python
def calculate_order_params(ticker):
    # READ (no writes)
    atr = MARKET_STATE[ticker]['atr']
    resistance = MARKET_STATE[ticker]['resistance']
```

Reader 3: Monitoring Dashboard (hypothetical)

```python
def get_dashboard_data():
    # READ (no writes)
    return {ticker: state['rsi'] for ticker, state in MARKET_STATE.items()}
```

Why This Matters - Lock-Free Performance:

Without SWMR (multiple writers):

```python
# Multiple threads writing
lock = threading.Lock()

def update_rsi(ticker, rsi):
    with lock:  # Serializes ALL access
        MARKET_STATE[ticker]['rsi'] = rsi

def read_rsi(ticker):
    with lock:  # Even READS must wait for lock
        return MARKET_STATE[ticker]['rsi']

# Performance:
# 1000 reads/sec × 1μs lock overhead = 1ms wasted per second
```

With SWMR (our approach):

```python
# No locks needed for reads
def read_rsi(ticker):
    return MARKET_STATE[ticker]['rsi']  # Direct access, 10ns

# Performance:
# 1000 reads/sec × 10ns = 10μs total (100x faster)
```

Enforcing SWMR Discipline:

1. **Code Review:** Verify only scheduler writes to MARKET_STATE
2. **Naming Convention:** `update_market_state()` only in scheduler module
3. **Immutability Practice:** Readers treat state as read-only

Violation Example (caught in code review):

```python
# websocket.py - BAD! Reader trying to write
def on_tick(ticker, ltp):
    MARKET_STATE[ticker]['last_ltp'] = ltp  # ❌ VIOLATION
```

This breaks SWMR—now WebSocket and Scheduler both write. Must refactor to have scheduler compute LTP-based features.

Concurrency Guarantees:

Python dict reads are atomic for built-in types:

```python
# Thread 1 (Writer)
MARKET_STATE['RELIANCE'] = new_state  # Atomic pointer assignment

# Thread 2 (Reader)
state = MARKET_STATE['RELIANCE']  # Atomic pointer read
```

Reader gets either old state or new state, never half-updated (torn read). This is why SWMR works without locks in Python.

Trade-offs:

- **Pro:** Lock-free reads, 100x faster
- **Con:** Must enforce discipline (no reader writes)

- **Pro:** No deadlocks possible (only one writer)
- **Con:** If we need multiple writers later, requires refactor

When We'd Violate SWMR: If we added real-time position updates:

```python
# execution.py now needs to write positions
MARKET_STATE[ticker]['position'] = {'qty': 100, 'entry': 1234}
```

Now have two writers (scheduler + execution). Would need:

- Separate dict for positions (POSITIONS_STATE)
- Or locks if they must share same dict
- Or message queue for writes

But for now, SWMR keeps it simple and fast.

Page 11: Shared-Nothing Across Processes, Shared-Everything Inside Process

First Principles

The unit of memory isolation is the process. Within a process, threads share memory naturally and efficiently. Across processes, sharing requires OS intervention (IPC) which is orders of magnitude slower. Optimal design respects this boundary: share within processes, isolate across processes.

Definition

Shared-Nothing Architecture (across processes) means each process has completely independent resources—no shared memory, files, or state. Communication happens via explicit message passing.

Shared-Everything Architecture (inside process) means threads share the same heap memory, can access same variables directly without copying.

This principle recognizes the natural boundary at the process level.

Layman's Explanation

Think of a company with multiple offices:

Across Offices (Processes): Can't walk to another office and grab their stapler. Must call/email (explicit communication).

Within Office (Threads): All employees share the same supply closet, meeting rooms, whiteboards. No formal requests needed.

Similarly:

- Across Python processes: Must use Redis/sockets
- Within one Python process: Just access same variable

Example 1: Web Server Architecture

Shared-Nothing (Nginx + workers):

```
Nginx spawns 4 worker processes
Each handles separate requests
No shared memory between workers
Load balancer routes requests randomly
```

Scales horizontally but can't share session state—need Redis.

Shared-Everything (Threads):

```
Flask app with 4 threads
All share same Python global variables
Session cache in RAM accessible to all threads
```

Doesn't scale across machines but simpler for single-server.

Example 2: Database Connection Pooling

Shared-Nothing:

```python
# Multiple processes
process1 → connection_pool1 (5 connections)
process2 → connection_pool2 (5 connections)
Total: 10 DB connections
```

Shared-Everything:

```python
```

```
# Multiple threads in one process
thread1 → shared_connection_pool (5 connections)
thread2 → shared_connection_pool (same pool)
Total: 5 DB connections (more efficient)
```

Application in Our Trading Bot

Design Decision: Single Process, Multiple Threads

Our Architecture:

```python
# main.py - ONE process
import threading
from state import MARKET_STATE  # Shared heap memory

def scheduler_thread():
    while True:
        MARKET_STATE['RELIANCE'] = compute_features()  # Direct write

def websocket_thread():
    while True:
        support = MARKET_STATE['RELIANCE']['support']  # Direct read

# Both threads in same process → same MARKET_STATE object
threading.Thread(target=scheduler_thread).start()
threading.Thread(target=websocket_thread).start()
```

Memory Access Cost:

```
Same process, different threads:
- Read MARKET_STATE: 10ns (L1 cache hit)
- No copying, no serialization

Different processes:
- Redis GET: 500µs (network + deserialization)
- 50,000x slower
```

Why We Chose Single-Process:

1. **Latency Requirements:**
    - Tick processing: 1000/sec → 1ms budget per tick

- Redis adds 500µs → would consume 50% of budget
- Shared memory: 10ns → negligible

2. **State Size:**
   - 5 tickers × 50 features × 8 bytes = 2KB
   - Fits in L1 cache (32KB)
   - No memory pressure
3. **Complexity:**
   - Single process: Import and use
   - Multi-process: Redis setup, serialization, error handling

Respecting Process Boundaries:

We understand the constraint:

```python
# This DOES NOT WORK (different processes)
# scheduler.py
from state import MARKET_STATE
MARKET_STATE['X'] = 1

# websocket.py (separate process)
from state import MARKET_STATE
print(MARKET_STATE['X'])  # KeyError - separate memory!
```

So we keep everything in one process:

```python
# main.py (ONE process)
from scheduler import run_scheduler
from websocket import run_websocket
# Both import same MARKET_STATE → works perfectly
```

When We'd Go Multi-Process:

Scenario: Scale to 100 tickers, 3 strategies, 5 execution venues

New Architecture:

```
Process 1: Data pipeline → writes to Redis
Process 2: Strategy 1 → reads Redis
Process 3: Strategy 2 → reads Redis
Process 4: Execution → reads Redis, writes orders
```

Now shared-nothing across processes, Redis as message bus.

Trade-offs We Accept:

Pros (Single Process):

- 50,000x faster state access
- No serialization overhead
- Simpler code, easier debugging
- One process to monitor/restart

Cons (Single Process):

- Can't scale horizontally
- Single point of failure
- Memory limit to one machine

For 5 tickers on one EC2 instance, pros massively outweigh cons.

Advanced: Why Not Multiprocessing with Shared Memory?

Python's `multiprocessing.Manager` provides shared memory:

```python
from multiprocessing import Manager
manager = Manager()
shared_state = manager.dict()  # Shared across processes
```

But it's slow:

```
Threading (our approach): 10ns access
Multiprocessing.Manager: 100µs (IPC overhead)
Redis: 500µs (network overhead)
```

Manager is 10,000x slower than threads. Only use if need process isolation for stability, not performance.

Page 12: Replace, Don't Mutate (Atomic State Transitions)

First Principles

In concurrent systems, partial updates to shared state create race conditions where readers see inconsistent data. By replacing entire state objects atomically rather than mutating fields individually, we ensure readers always see coherent snapshots—never half-updated state.

Definition

Replace, Don't Mutate (also called **Atomic State Transitions**) is a concurrency pattern where instead of modifying an object's fields in place, you create a new object with updated values and atomically replace the old reference with the new one. This ensures all-or-nothing visibility to concurrent readers.

Layman's Explanation

Imagine editing a document:

Mutate (dangerous): Open the file, edit paragraph 1, edit paragraph 2, save. Someone reading while you edit sees half-updated document.

Replace (safe): Save current version as "v1.pdf". Create new file "v2.pdf" with all edits. When done, rename "v2.pdf" to "current.pdf". Readers see either v1 (complete) or v2 (complete), never half-edited.

Similarly, instead of updating RSI then ATR then support (three separate mutations), we create new state object with all fields, then swap it in.

Example 1: React State Updates

Bad (mutating):

```javascript
// Reader might see partial update
this.state.user.name = "Alice";
this.state.user.age = 30;
```

Good (replacing):

```javascript
// Reader sees old state or new state, never torn
this.setState({
  user: { name: "Alice", age: 30 }  // New object
});
```

React enforces immutability for this reason.

Example 2: Database Transactions

Bad (non-atomic):

```sql
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
-- Reader queries here → sees inconsistent state
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
```

Good (atomic transaction):

```sql
BEGIN TRANSACTION;
  UPDATE accounts SET balance = balance - 100 WHERE id = 1;
  UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;  -- Atomically visible
```

Readers see pre-transaction or post-transaction, never intermediate.

Application in Our Trading Bot

Our Pattern: Replace Entire Dict

Atomic (our approach):

```python
# scheduler.py
def pipeline_function(ticker):
    # Compute ALL features FIRST (local variables)
    features = {
        'support': 1234.50,
        'resistance': 1250.00,
        'rsi': 45.2,
        'atr': 12.5,
        'ma_50': 1235.0,
        'ml_prediction': 1,
        'ml_confidence': 0.75
    }

    # Single atomic replacement
    MARKET_STATE[ticker] = features  # ← Atomic pointer swap
```

Non-Atomic (anti-pattern):

```python
# DON'T DO THIS
def pipeline_function(ticker):
    MARKET_STATE[ticker]['support'] = compute_support()      # Update 1
    MARKET_STATE[ticker]['resistance'] = compute_resistance()  # Update 2
    MARKET_STATE[ticker]['rsi'] = compute_rsi()              # Update 3
    # ... 5 more updates

    # Reader accessing between updates sees inconsistent state:
    # - support from new calculation
    # - resistance from old calculation
```

Race Condition Example:

Timeline with Mutation:

```
T0: MARKET_STATE['RELIANCE'] = {'support': 1200, 'rsi': 40, 'ml_pred': 1}
T1: Scheduler updates support: {'support': 1234, 'rsi': 40, 'ml_pred': 1}
T2: WebSocket reads → support=1234, rsi=40, ml_pred=1  (inconsistent!)
T3: Scheduler updates rsi: {'support': 1234, 'rsi': 45, 'ml_pred': 1}
T4: Scheduler updates ml_pred: {'support': 1234, 'rsi': 45, 'ml_pred': -1}
```

At T2, WebSocket sees new support but old RSI/prediction—might trigger trade with wrong signals.

Timeline with Replacement:

```
T0: MARKET_STATE['RELIANCE'] = {'support': 1200, 'rsi': 40, 'ml_pred': 1}
T1-T3: Scheduler computes locally (MARKET_STATE unchanged)
T4: Scheduler replaces: MARKET_STATE['RELIANCE'] = {'support': 1234, 'rsi': 45, 'ml_pred': -1}
T5: WebSocket reads → support=1234, rsi=45, ml_pred=-1  (consistent!)
```

WebSocket sees either old snapshot (T0-T3) or new snapshot (T4+), never mixed.

Why This Works - Python Guarantees:

Python dict assignment is atomic:

```python
```

```
# CPython implementation (simplified)
dict[key] = value
↓
PyDict_SetItem(dict, key, value) {
    hash = hash(key);
    entry = find_entry(hash);
    old_value = entry->value;
    entry->value = value;  // ← Single pointer write (atomic on x86/ARM)
    DECREF(old_value);
}
```

Single pointer write is atomic at CPU level. No torn reads possible.

Contrast with Field Mutation:

```python
# NOT atomic across fields
state['support'] = 1234    # Write 1
state['rsi'] = 45          # Write 2
state['ml_pred'] = -1      # Write 3

# Reader can see any combination:
# (1234, 40, 1) - saw write 1
# (1234, 45, 1) - saw writes 1-2
# (1234, 45, -1) - saw all writes
```

Benefits Realized:

1. **Consistency:** WebSocket never sees partial updates
2. **No Locks Needed:** Atomic replacement works without synchronization
3. **Debuggability:** Each snapshot is coherent—can log and reason about
4. **Correctness:** Impossible to trade on (new support, old ML prediction) combination

When Mutation is Safe:

Single-field updates can be atomic:

```python
MARKET_STATE[ticker]['last_tick_time'] = now()  # One write, atomic
```

But multi-field updates require replacement:

```python
# Atomic
MARKET_STATE[ticker] = {'support': X, 'rsi': Y, 'ml': Z}

# Not atomic
MARKET_STATE[ticker]['support'] = X
MARKET_STATE[ticker]['rsi'] = Y
MARKET_STATE[ticker]['ml'] = Z
```

Advanced: Copy-on-Write Alternative:

Some systems use COW:

```python
# Create modified copy
new_state = MARKET_STATE[ticker].copy()
new_state['support'] = 1234
new_state['rsi'] = 45

# Atomic replacement
MARKET_STATE[ticker] = new_state
```

We skip COW because we compute all features in one pass anyway—no existing state to copy.

Page 13: Temporal Determinism

First Principles

In systems that make decisions based on time-series data, the temporal consistency of inputs determines output correctness. If features are computed from different time windows or updated asynchronously, decisions become non-reproducible. By aligning all feature calculations to consistent time boundaries and separating time-driven computation from event-driven observation, we ensure deterministic, backtestable behavior.

Definition

Temporal Determinism is a system design principle for time-series systems where:

1. Features are computed on fixed time boundaries (e.g., 5-minute candles)
2. All features for a given timestamp use data from the same time window
3. Event-driven components (e.g., tick handlers) observe snapshots from these boundaries
4. Given the same historical data, the system produces identical outputs

This separates **signal formation** (time-based, deterministic) from **signal observation** (event-based, reactive).

Layman's Explanation

Imagine a weather forecast:

Non-Deterministic: Check temperature every 5 seconds, humidity every 7 seconds, pressure every 11 seconds. When you make a prediction, each input is from a different time—impossible to reproduce.

Deterministic: Every hour on the hour, measure temperature, humidity, and pressure simultaneously. Predictions use synchronized snapshots. Given historical snapshots, predictions are reproducible.

Our bot measures RSI, support, MAs all from the same 5-minute candle boundary, not from random moments when ticks arrive.

Example 1: Financial Backtesting

Bad (non-deterministic):

```python
# Live trading
rsi = calculate_rsi(latest_ticks)  # Uses last 100 ticks, whenever they arrived

# Backtesting
rsi = calculate_rsi(historical_candles)  # Uses 5-min candles

# Different inputs → different RSI → can't validate strategy
```

Good (deterministic):

```python
# Both live and backtest
candle = aggregate_ticks_to_5min(data)  # Same time boundary
rsi = calculate_rsi(candle)  # Same inputs → same RSI
```

Example 2: Scientific Experiments

Bad: Measure plant growth at "various times during the day" → Impossible to reproduce

Good: Measure every day at 9:00 AM exactly → Reproducible protocol

Application in Our Trading Bot

Our Temporal Architecture:

Time-Driven Layer (Deterministic):

```python
# Runs every 5 minutes on the minute (09:15, 09:20, 09:25...)
@scheduler.scheduled_job('interval', minutes=5)
def pipeline_function(ticker):
    # Fetch candle aligned to 5-min boundary
    candle = fetch_latest_5min_candle(ticker)

    # Compute ALL features from SAME candle
    features = {
        'support': candle.rolling(10).min(),      # ← Same time window
        'rsi': calculate_rsi(candle),             # ← Same time window
        'ma_50': candle.rolling(50).mean(),       # ← Same time window
        'ml_prediction': model.predict(candle)    # ← Same time window
    }

    # Store snapshot with timestamp
    MARKET_STATE[ticker] = {
        **features,
        'timestamp': candle.index[-1]  # Exact time of snapshot
    }
```

Event-Driven Layer (Reactive, Non-Deterministic):

```python
# Fires on every tick (irregular timing)
def on_tick(ticker, ltp, tick_time):
    # Read SNAPSHOT from last 5-min boundary
    snapshot = MARKET_STATE[ticker]

    # Snapshot is deterministic (computed at T=09:20:00)
    # Tick time is non-deterministic (could be T=09:20:37.428)

    if ltp <= snapshot['support']:
        # Decision uses deterministic features
        # Even though trigger time is non-deterministic
        check_conditions(ticker, snapshot)
```

Key Insight: Separation of Concerns

Aspect	Time-Driven (Features)	Event-Driven (Signals)
Trigger	Clock (5-min intervals)	Market ticks (irregular)
Purpose	Compute features	Detect conditions
Determinism	Fully deterministic	Non-deterministic timing
Reproducibility	Can replay exactly	Can only approximate

Why This Matters - Backtesting:

```python
# Backtesting (historical data)
for timestamp in historical_5min_candles:
    features = compute_features(candles_up_to[timestamp])

    # Simulate ticks within this 5-min window
    for tick in ticks_within_5min[timestamp]:
        if tick.ltp <= features['support']:
            if all_conditions_met(features):
                record_trade(tick)

# EXACT same features as live (same time boundaries)
# → Backtest results are valid predictors of live performance
```

Non-Deterministic Approach (anti-pattern):

```python
# Compute features on every tick
def on_tick(ticker, ltp):
    # BAD: Features computed from "last 50 ticks"
    # → Different ticks in live vs backtest
    ma_50 = recent_ticks[-50:].mean()

    # BAD: Features computed "now" (wall clock time)
    # → Different data available in live vs backtest
    rsi = calculate_rsi(all_data_until_now())
```

Temporal Consistency Example:

9:20:00 - Candle Close:

```python
MARKET_STATE['RELIANCE'] = {
    'support': 1234.50,    # Computed from candles 8:15-9:20
    'rsi': 45.2,           # Computed from candles 8:15-9:20
    'ma_50': 1235.0,       # Computed from last 50 candles ending 9:20
    'ml_prediction': 1,    # Model trained on features ending 9:20
    'timestamp': '2026-02-03 09:20:00'
}
```

All features aligned to SAME moment (9:20:00). Can reproduce in backtest.

9:20:00 to 9:25:00 - Ticks arrive:

```
9:20:03.142 → LTP=1235.20 → Read snapshot from 9:20:00
9:20:15.891 → LTP=1234.10 → Read snapshot from 9:20:00
9:21:47.238 → LTP=1233.90 → Read snapshot from 9:20:00 (still)
...
```

All ticks during this window use the SAME feature snapshot. Tick timing doesn't affect features.

9:25:00 - New Candle:

```python
MARKET_STATE['RELIANCE'] = {
    'support': 1233.80,    # NOW uses candles 8:20-9:25
    'rsi': 46.1,           # NOW uses candles 8:20-9:25
    ...
}
```

Features update atomically at time boundary.

Benefits Realized:

1. **Backtestability:** Replay historical candles → get exact same features
2. **Reproducibility:** Given same input data, always same output
3. **Debuggability:** Can inspect exact feature state at any 5-min boundary
4. **Correctness:** No lookahead bias (features only use data available at that time)
5. **Compliance:** Can prove to regulators what data was used for each decision

Trade-offs:

Pro: Complete determinism of decision inputs
Con: Features lag by up to 5 minutes (using snapshot from last boundary)

Why Lag is Acceptable:

- RSI/MA/support don't change drastically in 5 minutes
- Trading on slightly stale features (5 min old) better than non-reproducible features
- Backtests already assume 5-min granularity—live system must match

Advanced: Event Time vs Processing Time

We use **event time** (when market event occurred) not **processing time** (when our system processed it):

```python
# Event time (correct)
candle_timestamp = candle.index[-1]  # When candle closed in market

# Processing time (wrong)
processing_timestamp = datetime.now()  # When our code ran
```

Even if network delay means we process 9:20:00 candle at 9:20:03, we timestamp it as 9:20:00.

This is critical for determinism—backtests use event time, live must match.

Page 14: YAGNI (You Aren't Gonna Need It)

First Principles

Every abstraction, framework, and infrastructure component adds complexity—code to write, systems to maintain, failure modes to handle. Premature optimization and over-engineering waste resources solving problems you don't have. The optimal design for current requirements is simpler than the optimal design for hypothetical future requirements.

Definition

YAGNI (You Aren't Gonna Need It) is an Extreme Programming principle stating: don't add functionality, infrastructure, or abstraction until you have concrete evidence you need it. Design for today's requirements with an eye toward future evolution, but don't build tomorrow's features today.

Layman's Explanation

Building a house:

YAGNI Violator: "We might have kids someday, so let's build 5 bedrooms, even though we're a couple." → Wasted money, harder to maintain.

YAGNI Practitioner: "Build 2 bedrooms now. If we have kids, we'll extend." → Right-sized for today, extensible for tomorrow.

Similarly, don't deploy Kubernetes for 5 stocks when a single Python script suffices.

Example 1: Microservices

Premature (violates YAGNI):

```
For a 1000-user app:
- User service (Docker container)
- Auth service (Docker container)
- API gateway (Nginx)
- Service mesh (Istio)
- Message queue (Kafka)
- Orchestration (Kubernetes)
```

Complexity: 1000+ hours setup, 10 systems to monitor.

YAGNI Approach:

```
Single Django monolith
- Runs on one server
- All features in one codebase
```

Complexity: 10 hours setup, 1 system to monitor.

When you hit 100,000 users, then split into microservices.

Example 2: Database Choice

Premature: "We might need to scale to billions of records, so let's use Cassandra from day 1." → 2 weeks learning Cassandra, eventual consistency bugs

YAGNI: "We have 10,000 records. Use PostgreSQL." → 1 hour setup, works perfectly

When you hit 10M records, migrate. Don't solve tomorrow's problem today.

Application in Our Trading Bot

Our YAGNI Decisions:

Decision 1: RAM vs Redis

Could Have Done (violates YAGNI):

```python
# "We might scale to 500 tickers across 10 servers someday"
import redis
r = redis.Redis()

def update_state(ticker, features):
    r.set(f'state:{ticker}', json.dumps(features))

def get_state(ticker):
    return json.loads(r.get(f'state:{ticker}'))
```

Cost:

- 2 days Redis setup, configuration, monitoring
- 500µs latency per read (vs 10ns for dict)
- Network dependency (another failure mode)
- Serialization overhead

What We Did (YAGNI):

```python
# "We have 5 tickers on 1 server today"
MARKET_STATE = {}

MARKET_STATE[ticker] = features  # 10ns
state = MARKET_STATE[ticker]     # 10ns
```

Cost: 5 minutes, zero infrastructure

Decision Point: When we scale to 50+ tickers or multiple servers, then add Redis. Not before.

Decision 2: Single Process vs Microservices

Could Have Done:

```
Scheduler service (Docker)
WebSocket service (Docker)
Execution service (Docker)
Message queue (RabbitMQ)
API gateway (Nginx)
Orchestration (Docker Compose)
```

Complexity: 1 week setup, 6 systems, distributed tracing needed

What We Did:

```python
# main.py - one process
threading.Thread(target=run_scheduler).start()
threading.Thread(target=run_websocket).start()
```

Complexity: 10 lines, 1 system

When We'd Change: If we need separate deployment cycles for components, then containerize.

Decision 3: Configuration Management

Could Have Done:

```
- HashiCorp Consul (service discovery)
- Environment-specific configs (dev/staging/prod)
- Secret management (Vault)
- Feature flags (LaunchDarkly)
```

What We Did:

```python
# config.py
TICKERS = ["RELIANCE", "INFY", "ICICIBANK", "ADANIPORTS", "HINDALCO"]
CONFIDENCE_THRESHOLD = 0.6
```

5 tickers, one environment (prod). Hard-code it.

When we have 100 tickers, then use config files. When we have staging environment, then externalize configs.

Decision 4: Monitoring & Observability

Could Have Done:

```
- Prometheus (metrics)
- Grafana (dashboards)
- ELK stack (logging)
- Jaeger (distributed tracing)
- PagerDuty (alerting)
```

Setup Cost: 2 weeks, 5 systems

What We Did:

```python
import logging
logging.info(f"{ticker} | Pred: {pred} | Confidence: {conf}")
```

Plus systemd for auto-restart. Good enough for 5 stocks.

When We'd Add: If managing 50+ strategies or running at quant fund scale, then full observability stack.

Decision 5: Testing Infrastructure

Could Have Done:

```
- Unit tests (pytest)
- Integration tests (testcontainers)
- E2E tests (Selenium)
- Load tests (Locust)
- CI/CD (GitHub Actions)
```

What We Did:

- Walk-forward validation of ML models (already done)
- Manual testing of order placement
- Backtest against historical data

When We'd Add: When multiple developers, then CI/CD. When strategy changes frequently, then automated regression tests.

Benefits of YAGNI in Our Project:

1. **Time to Market:** Live trading in weeks, not months
2. **Simplicity:** Can understand entire system in one sitting
3. **Debugging:** One log file, one process, one server
4. **Cost:** $10/month EC2 vs $500/month infrastructure
5. **Velocity:** Changes take minutes, not coordination across services

What We're Ready to Add (When Needed):

Trigger	Solution
Scale to 50+ tickers	Add Redis
Multiple strategies	Separate services
Team of 3+ developers	CI/CD pipeline
Production incidents	Full monitoring stack
Regulatory audit	Complete logging infrastructure

YAGNI is Not:

- Ignoring future needs
- Writing unmaintainable code
- Avoiding design altogether

YAGNI is:

- Building for **today's requirements**
- Designing for **easy evolution**
- Adding complexity only when **evidence demands it**

Our Evidence Threshold:

We'll add infrastructure when we have concrete data:

- Redis: When dict access becomes CPU bottleneck (profiler shows it)
- Multiple processes: When one crashes and we need isolation
- Queue: When producer/consumer rates mismatch causes problems

Not based on "we might need it" but "we're experiencing pain without it."

Page 15: Fail-Fast, Debuggable Systems

First Principles

System complexity grows exponentially with the number of interacting components. Distributed systems (multiple processes, machines, networks) multiply failure modes and obscure causality. For domains requiring deterministic behavior and rapid iteration, a monolithic architecture with centralized logging and clear failure points enables faster debugging than distributed systems.

Definition

Fail-Fast Systems crash loudly and immediately when encountering errors, rather than masking failures or entering undefined states.

Debuggable Systems provide observability through:

1. Single point of logging (not scattered across services)
2. Reproducible execution paths
3. Ability to attach debuggers
4. Clear causality chains (A caused B caused C)

For early-stage systems, these properties often outweigh the scalability benefits of distributed architectures.

Layman's Explanation

Distributed System (hard to debug): "Car makes strange noise. Could be engine (process 1), transmission (process 2), or suspension (process 3). Each has separate diagnostics. Noise happens intermittently across all three."

Monolithic System (easy to debug): "Bike makes strange noise. Open up, see it's the chain. Fix chain. Done."

Similarly, when your trading bot fails, you want one log file showing exactly what happened, not 5 services with interleaved timestamps across network delays.

Example 1: Microservices Debugging Nightmare

Scenario: Order fails to execute.

Distributed System:

```
1. Check API gateway logs
2. Check auth service logs (different server)
3. Check order service logs (different server)
4. Check database logs
5. Correlate timestamps across 4 systems
6. Realize network delay between services caused timeout
7. 2 hours debugging
```

Monolithic System:

```
1. Check application.log
2. See: "Order validation failed: Insufficient capital"
3. 2 minutes debugging
```

Example 2: Race Condition Debugging

Distributed:

```
Service A: 09:20:00.123 - Updating state
Service B: 09:20:00.125 - Reading state
Service C: 09:20:00.127 - Reading state

Which service saw which version of state?
Need distributed tracing, clock sync, correlation IDs.
```

Monolithic:

```
Thread 1: 09:20:00.123 - Updating state
Thread 2: 09:20:00.125 - Reading state

Same process, same clock, sequential log entries.
Trivial to understand causality.
```

Application in Our Trading Bot

Why We Chose Monolithic Architecture:

Single Log File:

```python
```

```
# All modules log to same file
import logging
logging.basicConfig(
    filename='trading_bot.log',
    format='%(asctime)s | %(module)s | %(message)s'
)

# Output
2026-02-03 09:20:00.123 | scheduler | RELIANCE features updated
2026-02-03 09:20:00.456 | websocket | RELIANCE tick received: 1234.50
2026-02-03 09:20:00.789 | websocket | Conditions met for RELIANCE
2026-02-03 09:20:01.012 | execution | Order placed: RELIANCE BUY 100
```

Complete Causality Chain in One File:

- See feature update
- See tick that triggered condition
- See condition check
- See order placement
- All with <1ms timestamps

Contrast with Distributed:

```
scheduler.log:    2026-02-03 09:20:00.123 | Features updated
websocket.log:    2026-02-03 09:20:00.456 | Tick received
execution.log:    2026-02-03 09:20:01.012 | Order placed

Where's the condition check log?
In which service?
How do we correlate across 3 files?
```

Fail-Fast Example:

```
python
```

```python
def execute_trade(ticker):
    # Check 1: Position already open?
    if has_open_position(ticker):
        raise RuntimeError(f"Position already exists for {ticker}")

    # Check 2: Sufficient capital?
    if get_available_capital() < required_capital:
        raise RuntimeError(f"Insufficient capital")

    # Check 3: Market hours?
    if not is_market_open():
        raise RuntimeError("Market closed")

    # If we reach here, all validations passed
    place_order(ticker)
```

Benefit: If ANY check fails, entire process crashes with clear error. We don't silently skip the trade or enter undefined state.

Single Process → Entire Bot Crashes:

```
2026-02-03 09:25:00.456 | execution | RuntimeError: Insufficient capital
Traceback (most recent call last):
  File "execution.py", line 45, in execute_trade
    raise RuntimeError("Insufficient capital")
```

Systemd restarts bot. We see exactly what failed, when, why.

In Distributed System: Execution service fails, but scheduler and websocket keep running. They're sending messages to dead service. Need health checks, circuit breakers, retry logic. 10x complexity.

Debugger Attachment:

Monolithic:

```bash
```

```
# Attach Python debugger to running process
python -m pdb main.py

# Set breakpoint
(Pdb) break websocket.py:45

# Step through live trading
(Pdb) step
```

Can inspect MARKET_STATE, thread state, everything in one process.

Distributed: Can't attach debugger across services. Need to reproduce issue in isolated service, losing context.

Reproducibility:

Scenario: Trade executed at wrong price on 2026-02-03 09:20:15.

Monolithic:

```python
# Replay from logs
1. Load historical ticks until 09:20:15
2. Run scheduler to rebuild MARKET_STATE
3. Feed exact tick sequence to websocket
4. Observe exact state at moment of execution
5. Find bug: RSI was actually 56, not 45 (logging error)
```

Distributed: Impossible to replay unless you recorded all inter-service messages with nanosecond timestamps and network delays.

Performance Observability:

Monolithic:

```python
```

```
import cProfile
cProfile.run('main.py')

# See exactly where time is spent:
- 40% in feature calculation
- 30% in SQLite queries
- 20% in ML inference
- 10% in WebSocket processing
```

Distributed: Need APM tools (Datadog, New Relic) costing $100/month to get similar visibility.

When Monolith Becomes a Liability:

Scenario 1: WebSocket connection unstable, crashes every hour

- Currently: Entire bot crashes, no trades during restart
- With services: WebSocket restarts, scheduler/execution unaffected

Scenario 2: ML inference takes 10 seconds on new model

- Currently: Blocks entire process
- With services: Runs in separate service, doesn't block ticks

Scenario 3: Need to deploy new ML model without restarting WebSocket

- Currently: Full restart, lose tick stream
- With services: Deploy ML service independently

Our Threshold: When operational pain from single-process outweighs debugging complexity of distributed.

For 5 tickers with <500ms processing, monolith is still optimal.

Trade-offs Summary:

Aspect	Monolith (Our Choice)	Distributed
Debugging	One log file, clear causality	Distributed tracing needed
Deployment	One binary, one command	Multiple services, orchestration
Scaling	Vertical only (bigger machine)	Horizontal (more machines)
Complexity	Low (1 system)	High (N systems)
Isolation	No isolation (shared failure)	Strong isolation
Latency	Nanoseconds (function call)	Milliseconds (network)
Best For	Early stage, tight iteration	Mature product, high scale

We're early stage with 5 tickers. Monolith is correct choice.

When we're at quant fund scale with 500 tickers and $10M AUM, we'll migrate to microservices with full observability. Not before.

Conclusion

These 14 principles form a coherent system design philosophy:

Architecture (1-6): How to structure components
Concurrency (7-12): How to share state safely
Discipline (13-14): When to add complexity

Together, they produce a system that is:

- **Fast** (lock-free, in-memory)
- **Correct** (deterministic, atomic)
- **Simple** (monolithic, right-sized)
- **Evolvable** (modular, not over-engineered)

This is professional-grade systems engineering applied to algorithmic trading—the foundation for scaling from 5 tickers to a full quant operation.