**Real-Time Bond Inventory Platform - Technical Design**

**Summary**

A high-performance web application handling 2,000+ bond price updates/sec with real-time UI updates, filtering, and analytics for 500K bonds,

A diagram of a software company

AI-generated content may be incorrect.

[link](https://www.mermaidchart.com/app/projects/d4894e98-4273-4eee-9564-d46cfa309ef2/diagrams/43037985-0ed1-4739-9bd5-33b366511e4c/share/invite/eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkb2N1bWVudElEIjoiNDMwMzc5ODUtMGVkMS00NzM5LTliZDUtMzNiMzY2NTExZTRjIiwiYWNjZXNzIjoiRWRpdCIsImlhdCI6MTc1MzIzNTA1MX0.8nzUP_KcmPLjpaW8BcwxXOyG86dRQkRBVqaVU0zBFeQ)

**Business Requirement → Technical Implementation**

| **Business Need** | **Technical Solution** | **Implementation Details** | **PROD Technology** |
| --- | --- | --- | --- |
| **2,000 updates/sec** | **Producer-consumer pipeline with batching** | **Queues between workers, batching 2,000 updates for WebSocket delivery.** | Kafka for scalable event |
| **Real-time UI updates** | **WebSocket with batch notifications** | **Batch updates with reception/re-try mechanisms** | **Angular with WS** |
| **500K bond inventory** | **In-memory caching with efficient lookup** | **Pre-allocated arrays, Dictionary for O(1) lookups** | **Redis for distributed coaching, SQL for Persistent** |
| **Filtering & sorting** | **REST API with query parameters** | **LINQ queries on cached data** | **AG-Grid for dynamic display and Virtual scrolling** |
| **User customization** | **Client-side state + persistent storage** | **Local storage API with persistent storage solution to maintain data** | **PostgreSQL for data storage** |
| **Analytics** | **aggregate statistics** | **Time series of compound data** | **historical analytics in TimescaleDB.** |
| **T&C integration** | **Merge with price updates** | **Fetch T&C from PostgreSQL, cache in Redis, join in StatCalculationWorker.** | **PostgreSQL and Redis** |

**Core Implementation Approach**

**System Architecture**

**Data Flow Pipeline**

External Provider → Ingestion Queue → Metrics Calculator (merges T&C)

→ Cache Update → Batch Processor → WebSocket → Frontend

**Core Components**

1. **DummyInventoryProvider**: Simulates external bond feed (2,000 updates/sec)
2. **StatCalculationWorker**: Computes bond metrics (YTM, Duration, Convexity)
3. **CacheUpdateWorker**: Maintains in-memory bond state with deduplication
4. **BatchNotificationWorker**: Batches updates (2,000/batch) with deduplication
5. **REST API**: Handles filtering, sorting, pagination queries
6. **WebSocket Server**: Pushes real-time updates to connected clients

**Technology Choices & Justification**

**Backend: .NET 8 with ASP.NET Core**

* **Why**: High-performance, excellent WebSocket support, strong typing for financial calculations
* **BlockingCollection**: Thread-safe producer-consumer queues for worker pipeline
* **Memory-efficient**: Pre-allocated arrays for hot bond data

**Frontend: JS framework with WebSockets**

* **Why:**  Would use Angular/React with virtual scrolling for production scale
* **Real-time Updates**: WebSocket integration with visual change indicators

**Data Storage Strategy**

Hot Data: In-memory arrays (500K bonds × 2KB ≈ 1GB RAM)

User Preferences: Would add PostgreSQL for production

Time Series: Would add TimescaleDB for historical analytics

**Implementation Details**

**Performance Optimizations**

**Batching Strategy**

// Batch 2,000 updates with deduplication

**Memory Management**

* JSON serialization only once per update
* Lock-free reads with occasional write locks
* **Production level**: Redis for distributed coaching, Kafka for scalable event streaming to handle 2,500+ updates/sec.

**API Design**

GET /api/bonds?page=1&size=100&search=AAPL&sortBy=yield&minPrice=900&maxPrice=1100

GET /api/bonds/summary // Aggregate statistics

WebSocket /ws // Real-time bond updates

**Monitoring & Observability**

**Production additions**: performance metrics, structured logging, health checks, circuit breakers

**Communication Patterns**

**1. External Provider → Kafka (Event Streaming)**

**Why Event Streaming:**

* **Durability: Can replay messages if processing fails**
* **Throughput: Handles 2,500+ msg/sec easily**
* **Ordering: Per-partition ordering (by bond ID)**
* **Multiple consumers: Can add analytics consumers later**
* **Backpressure handling: Messages queue safely if consumers slow down**

**2. Kafka → Stats Calculator (Queue Consumer Pattern)**

**Why Consumer Groups:**

* **Parallel processing: Multiple workers can process different partitions**
* **Automatic failover: If worker dies, another picks up its partition**
* **Offset management: Tracks which messages were processed**
* **At-least-once delivery: Guarantees message processing**

**3. Internal Communication (In-Process)**

**Pattern: Direct method calls and in-memory queues  
Components: Stats Calculator ↔ Cache ↔ Batch Processor**

**Why Internal Communication:**

* **Lowest latency: No serialization/network overhead**
* **Simple: Direct code flow**
* **Same process: Both components run in same application**
* **Thread-safe: Using BlockingCollection for producer-consumer patterns**

**Note: could use Service Bus and HTTP for micro services approach**

**4. WebSocket Gateway → Frontend (Real-time Protocol)**

**Pattern: Persistent connection streaming  
Delivery: Batches of 200 bonds**

**Why WebSocket:**

* **Real-time: Sub-second update delivery**
* **Efficient: Persistent connection, no HTTP overhead per message**
* **Bidirectional: Can handle client requests (ping/pong)**
* **Browser native: Built-in browser support**
* **Batching support: Can send multiple updates in a single message**

**5. Frontend → REST API (Request-Response)**

**Pattern: HTTP request-response  
Use case: Queries, filtering, pagination**

**Why REST:**

* **Standard pattern: Well-understood HTTP semantics**
* **Caching: HTTP caching headers for performance**
* **Stateless: Easy to scale horizontally**
* **Tooling: Excellent debugging and monitoring tools**

**Testing Strategy**

**Unit Tests**

* Bond metrics calculations (YTM, Duration accuracy), Filtering logic validation
* Batching and deduplication algorithms

**Integration Tests**

* End-to-end pipeline throughput testing
* WebSocket connection handling
* API pagination and filtering

**Performance Tests**

* Load testing: 5,000+ updates/sec sustained (2x max load)
* Memory leak detection under continuous operation
* Frontend responsiveness with large datasets

**Risk Mitigation**

**External Feed Reliability**

* **Circuit breaker** pattern for provider failures
* **Dead letter queue** for failed message processing
* **Heartbeat monitoring** with automatic reconnection

**Features Dependencies**

**Phase 1: Core Data Pipeline – Ingestion and Processing**

**Critical Path**

* **Feature: High-frequency data processing pipeline with real-time distribution**
* **Story: As a trader, I need the system to process 2,500 bond updates per second and calculate financial metrics (YTW, Duration, Convexity) in real-time so I can make informed trading decisions based on current market data**
* **Technical Implementation:** 
  + **Build producer-consumer pipeline services**
  + **Create StatCalculationWorker for bond metrics computation**
  + **Develop CacheUpdateWorker with thread-safe in-memory storage**
  + **Set up BatchNotificationWorker with WebSocket broadcasting**
* **Acceptance Criteria:** 
  + **System processes 2,500+ updates/sec sustained**
  + **End-to-end latency <500ms from provider to cache**
  + **Basic WebSocket connection delivers batched updates**
* **Dependencies: Integration endpoints with bond data provider (can be mocked initially)**

**Phase 2: REST API Controllers – Data Access Layer**

* **Feature: RESTful API endpoints for bond data querying and filtering**
* **Story: As a trader, I need to filter bonds by criteria (issuer, price range, yield) and sort by any column so I can quickly find bonds matching my investment strategy**
* **Technical Implementation:** 
  + **Build BondsController with GET /api/bonds endpoint**
  + **Implement query parameters: page, size, search, sortBy, sortDir, price/yield filters**
  + **Create pagination logic with totalCount/hasNext metadata**
  + **Develop /api/bonds/summary endpoint for aggregate statistics**
* **Acceptance Criteria:** 
  + **API responses <200ms for filtered queries**
  + **Supports pagination with configurable page sizes**
  + **Handles 500K bond dataset efficiently**
* **Dependencies: Phase 1 core pipeline and cache infrastructure**

**Phase 3: User Interface – MVP Frontend**

* **Feature: Dynamic real-time bond inventory dashboard with live updates**
* **Story: As a trader, I need a responsive web interface that shows live bond price changes with visual indicators, allows me to filter/sort the inventory, and saves my preferences so I can efficiently monitor market movements**
* **Technical Implementation:** 
  + **Build responsive frontend with real-time WebSocket integration**
  + **Implement dynamic table with sorting, filtering, and pagination controls**
  + **Create filter persistence using localStorage/sessionStorage**
* **Acceptance Criteria:** 
  + **Filter/sort operations complete within 200ms**
  + **User preferences persist across browser sessions**
  + **Works smoothly with 500+ bonds displayed simultaneously**
* **Dependencies: Can be started with Phase 2 API or mock data for parallel development**

**Phase 4: Production Hardening & Monitoring**

* **Feature: Enterprise-grade reliability, monitoring, and scalability features**
* **Story: As a system administrator, I need comprehensive monitoring, error handling, and auto-recovery capabilities so the trading platform maintains 99.9% uptime during critical market hours**
* **Technical Implementation:** 
  + **Add circuit breaker pattern for external provider failures**
  + **Implement structured logging with correlation IDs**
  + **Create health check endpoints (/health, /ready)**
  + **Add metrics collection and Grafana dashboards**
  + **Implement dead letter queues for failed message processing**
  + **Add connection pooling and load balancing for WebSocket connections**
* **Acceptance Criteria:** 
  + **Comprehensive monitoring dashboard with key performance metrics**
  + **Performance degrades gracefully under load spikes**