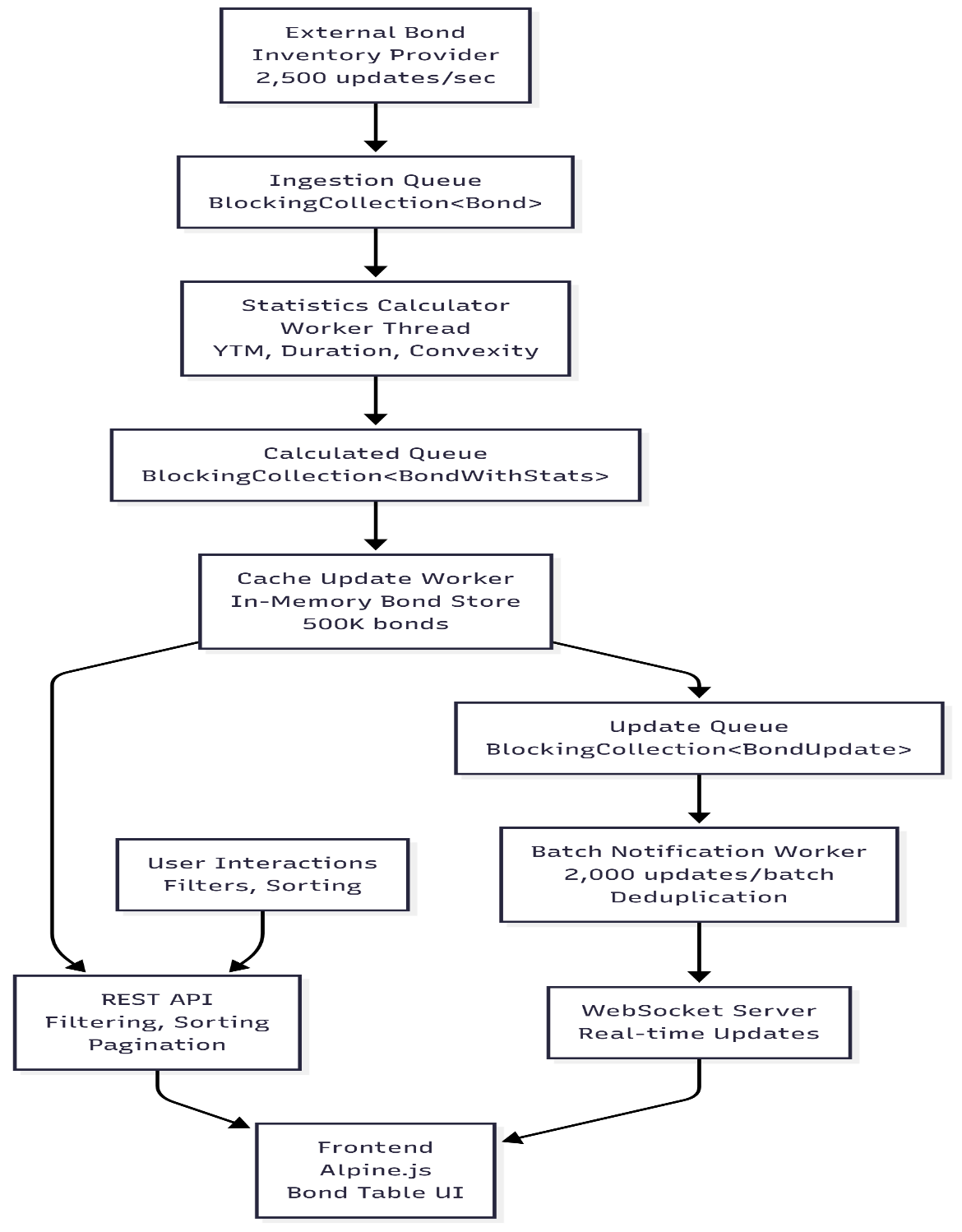
**Real-Time Bond Inventory Platform - Technical Design**

**Summary**

A high-performance web application handling 2,500 bond price updates/sec with real-time UI updates, filtering, and analytics for 500K bonds. Implements producer-consumer pipeline architecture with batched WebSocket updates.

[link](https://mermaid.live/edit#pako:eNqNk01v4jAQhv-K5UNPgSaEj5CuKvFZslVXbKFCu6EHk0wTL46NHAfBIv77Og5UIPbQHCJ7MvPM6zfjA45EDNjHS55IsknRfLjkSD-9cLRTIDlhqC94_G0l7x8DvgWuhNyjqRRbGoM04YbVsm1UbGKiIL_PIXpHtdoj6ocBTyBXVHD0s4ACTHKfiWhNeTIQjEFUfrxj6qFscZeoh_eqefXuG8wgnCmiqOZEORoQFhWMaA0GthByDRLNUwmkkvhr_mKhYSFJSbbQQGjJO6r2V-CBAQ_DMw3iLwlcUJWWWvIbpUMDHGlglAJ6M06gStzJuNoLZKVxJQfNtP6qmTbuGa10LL_ijQxvHJ5IXxFXpd4oGxvSU9gnKkrRD6HoB42MOZf6GpZ98QdXZa6JDyEuNuxU8B-Fk_B1NJuj3jQw6WPK9MhoeRaaCan0woSnJKH8FvFkEEG4gNVMnwoUmoHcnhS9AmE1RbOzm9f-TEzp93AsBVdwms4e21AO9T955VRp9JysmCYEp-KgKrskPYdvuZ6gQGMkMYbmF0fJPw9SjfTkXIotnEgaY1_JAiycgcxIucWHMmOJVQoZLLGvlzGR66W-XkddsyH8txDZuUyKIkmx_0FYrneV_UNK9EXMPqNSHw_kQBRcYb_Z8gwE-we8w36j6dSdVtNre12nYzfbTsfCe-zXnI5T95y20-p4Dbfddb2jhf-avk7d7dqO47btdqvj2q7bOv4DFqFFQA)

**Business Requirement → Technical Implementation**

| **Business Need** | **Technical Solution** | **Implementation Details** |
| --- | --- | --- |
| **2,500 updates/sec** | **Producer-consumer pipeline with batching** | **BlockingCollection<T> queues between workers** |
| **Real-time UI updates** | **WebSocket with batch notifications** | **Send 2,000 updates at once, client-side deduplication** |
| **500K bond inventory** | **In-memory caching with efficient lookup** | **Pre-allocated arrays, Dictionary for O(1) lookups** |
| **Filtering & sorting** | **REST API with query parameters** | **LINQ queries on cached data** |
| **User customization** | **Client-side state + persistent storage** | **Alpine.js for filters, future: database for saved preferences** |

**Core Implementation Approach**

**System Architecture**

**Data Flow Pipeline**

External Provider → Ingestion Queue → Metrics Calculator → Cache Update → Batch Processor → WebSocket → Frontend

**Core Components**

1. **DummyInventoryProvider**: Simulates external bond feed (2,500 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: Alpine.js with WebSockets**

* **Why**: Lightweight for POC, no build process, reactive data binding
* **Trade-off**: 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

Dictionary<string, int> bondIndicesById = new();

if (alreadyExists) {

currentBatch[index] = update.serializedStatus; // Replace, don't duplicate

}

**Memory Management**

* Pre-allocated fixed-size arrays (1M bonds max)
* JSON serialization only once per update
* Lock-free reads with occasional write locks

**Frontend Efficiency**

* Incremental DOM updates, not full re-renders
* Client-side price change tracking
* Debounced filter applications
* Visual update indicators with automatic timeout

**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**

// Performance tracking implemented

if (totalUpdates % 5000 == 0) {

float rate = 1000f \* totalUpdates / (DateTimeOffset.UtcNow.ToUnixTimeMilliseconds() - startTime);

Console.WriteLine($"Update rate: {rate:F1}/sec");

}

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

**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
* 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

**Data Consistency**

* **Optimistic locking** for concurrent price updates
* **Idempotent message processing**