Real-Time Price Aggregator: Performance Analysis Report

1. Performance Objectives

Our real-time price aggregator system aims to meet the following performance targets:

1.1 Latency

- P95 (95th percentile) GET request: < 80ms
- P99 (99th percentile) GET request: < 100ms

1.2 Throughput

- Peak GET requests: 1000+ requests/second

1.3 Availability

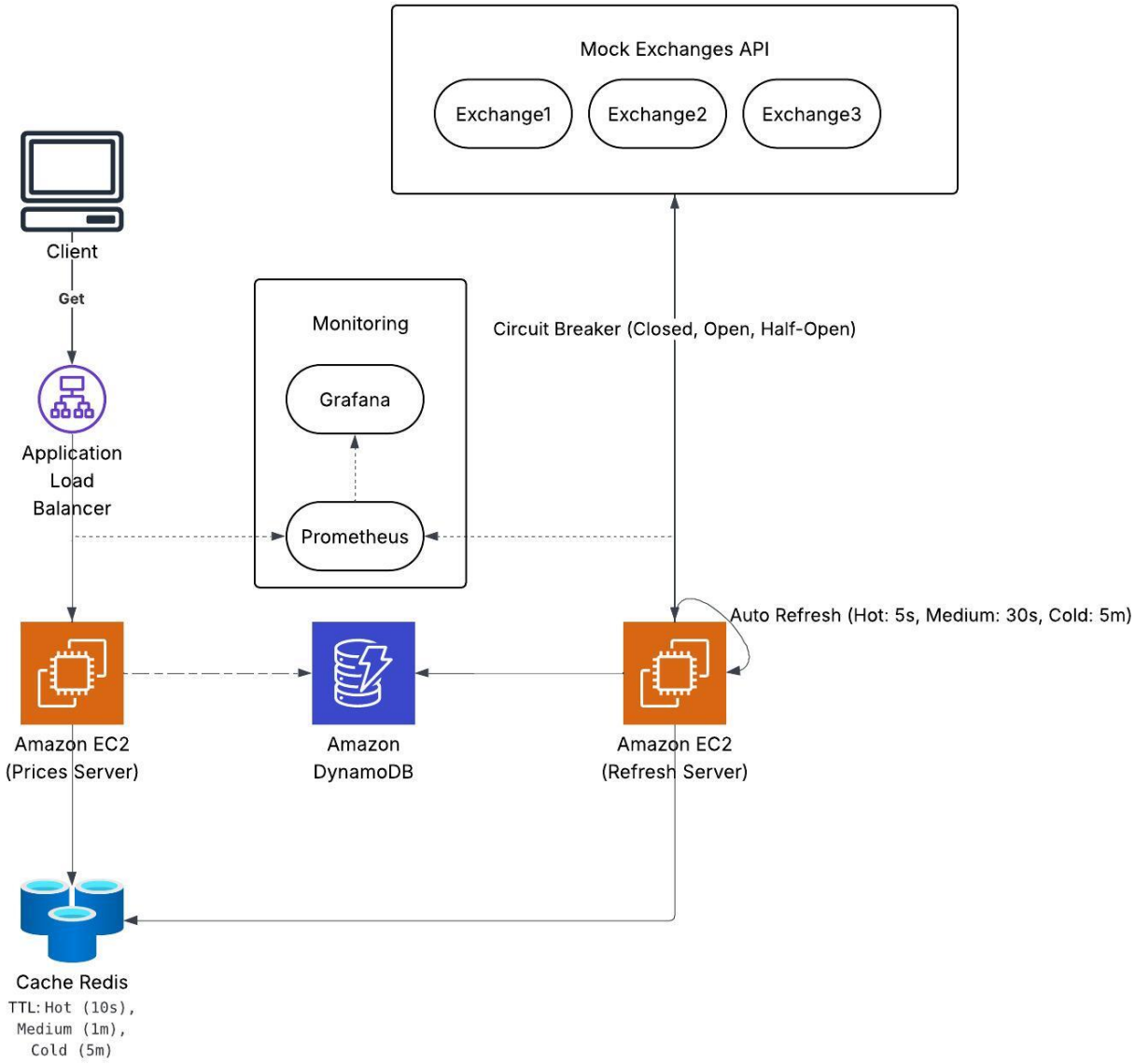
- API error rate: < 0.1%

1.4 Resource Utilization

- CPU usage: < 70%
- Memory usage: < 80%
- Redis cache hit rate: > 95%

2. System Architecture

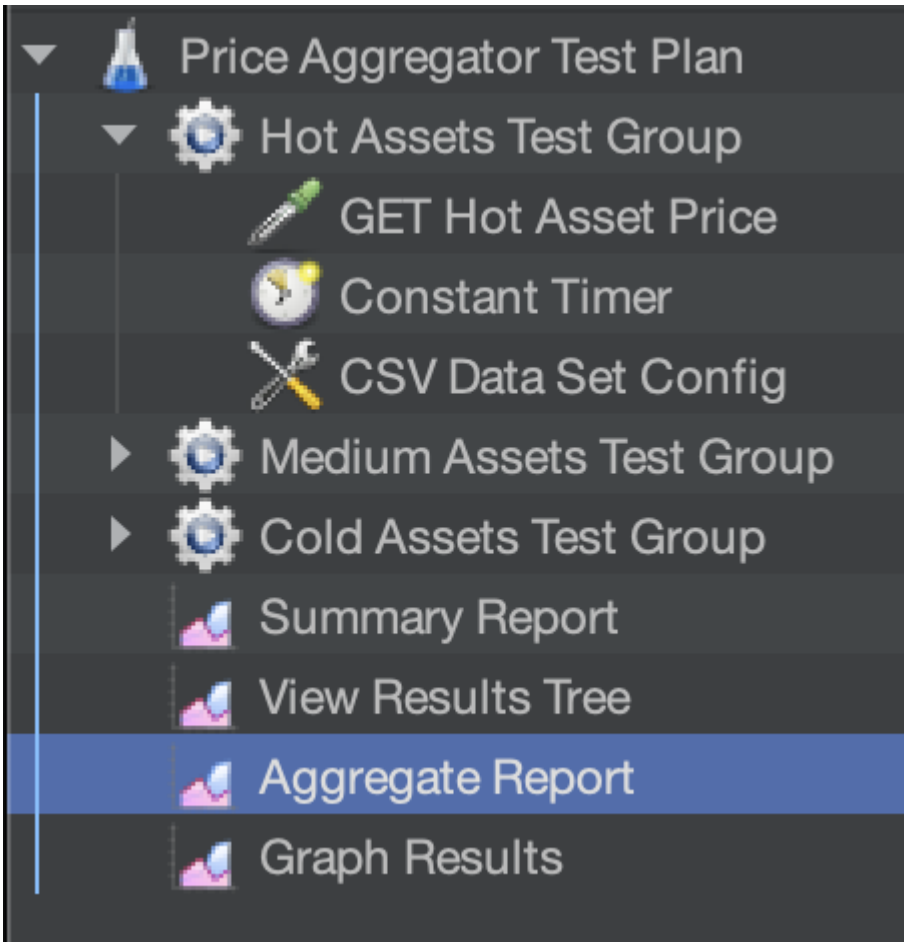
The current architecture implements a performance-optimized design with several key components:



- **Separated GET and POST Operations**: Implemented a CQRS-like pattern to separate read and write operations
- **Tiered Auto-Refresh Mechanism**: Implemented using Go's ticker for automatic price updates
  - Hot assets (top 10-20): Refresh every 5 seconds
  - Medium assets (next 100-200): Refresh every 30 seconds
  - Cold assets (remaining 201-1000): Refresh every 5 minutes
- **Circuit Breaker Pattern**: Enhanced error handling and system resilience
- **Monitoring**: Prometheus + Grafana for comprehensive system metrics
- **Cache Strategy**: Optimized Redis caching with tier-specific TTLs

3. Testing Methodology

Performance testing was conducted using JMeter with a tiered approach to simulate real-world traffic patterns:



- **Hot Assets Test Group**: Simulating high-frequency access to popular assets
- **Medium Assets Test Group**: Testing moderate traffic patterns
- **Cold Assets Test Group**: Evaluating less frequently accessed assets

4. Key Lessons Learned

An important testing methodology issue was discovered during the performance evaluation:

JMeter Results Accumulation: When executing multiple test runs without clearing previous results, JMeter accumulates data from previous test runs into current results. This initially led to misleading throughput measurements, where throughput appeared to decline from 1000+ requests/second to approximately 60 requests/second.

- Happens like this:
    - First test

Label	# Samp...	Average	Median	90% Li...	95% Li...	99% Li...	Min	Maxim...	Error %	Throug...	Receive...	Sent K...
GET H...	10000	29	24	49	67	92	14	150	0.00%	1645.8...	415.28	280.54
GET M...	6400	32	25	61	75	95	15	127	0.00%	1554.9...	397.98	266.57
GET C...	2400	35	27	68	78	98	15	124	0.00%	870.8/...	220.17	149.67
TOTAL	18800	31	25	56	73	95	14	150	0.00%	3094.1...	784.75	529.02

    - After a few tests

Label	# Samp...	Average	Median	90% Li...	95% Li...	99% Li...	Min	Maxim...	Error %	Throug...	Receive...	Sent K...
GET H...	42500	33	26	60	81	103	14	160	0.00%	97.2/sec	24.53	16.57
GET M...	22800	37	29	74	89	109	14	180	0.00%	52.4/sec	13.40	8.98
GET C...	7200	37	29	72	85	101	15	124	0.00%	16.6/sec	4.20	2.85
TOTAL	72500	35	27	66	85	105	14	180	0.00%	165.9/...	42.05	28.35

Solution: Before starting new tests, it's essential to:

1. Click the "Clear All" button in JMeter's toolbar
2. Use the menu: Run > Clear All
3. Or restart the JMeter application

This experience highlighted the importance of proper test setup and result management when conducting performance evaluations.

5. Performance Optimization Process

5.1 Initial API Integration Optimization

Early testing revealed performance limitations with the synchronous approach to external API calls. The system was modified to use concurrent processing:

- **Change**: Modified the API integration method from sequential synchronous calls to asynchronous execution with goroutines
- **Implementation**: Added goroutines in the `GetPrice` handler and explicitly used concurrent requests in the `FetchPrice` function
- **Result**: Significant throughput improvement (increased form 1000 req/s to 3094 req/s)

5.2 Cache Efficiency Optimization

Analysis of the cache performance metrics revealed high miss rates:

- **Problem**: All asset types (hot, medium, cold) were using the same cache expiration time
- **Solution**: Implemented tier-specific TTL values in the cache layer

```
// Based on asset tier, set different TTLs
var ttl time.Duration
switch tierType {
case "hot":
    ttl = 10 * time.Second  // Hot assets short TTL for freshness
case "medium":
    ttl = 1 * time.Minute   // Medium assets with moderate TTL
case "cold":
    ttl = 5 * time.Minute   // Cold assets with longer TTL to reduce DB
access
default:
    ttl = 5 * time.Minute
}
```

- **Result**: Cache miss rate dropped significantly, approaching zero

- Before and after comparison:

vs

5.3 Memory Management Optimization

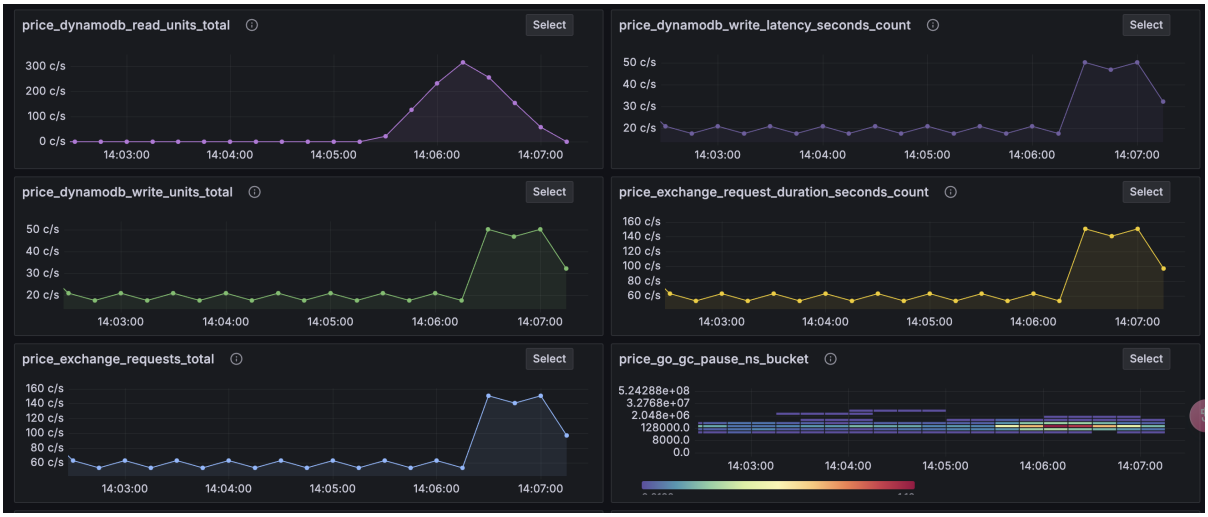
Testing revealed potential memory pressure during high load:

- **Problem**: Excessive temporary object allocation in `fetcher.go`
- **Solution**:
  - Implemented object pooling to reduce GC pressure
  - Adjusted GC parameters to optimize memory management

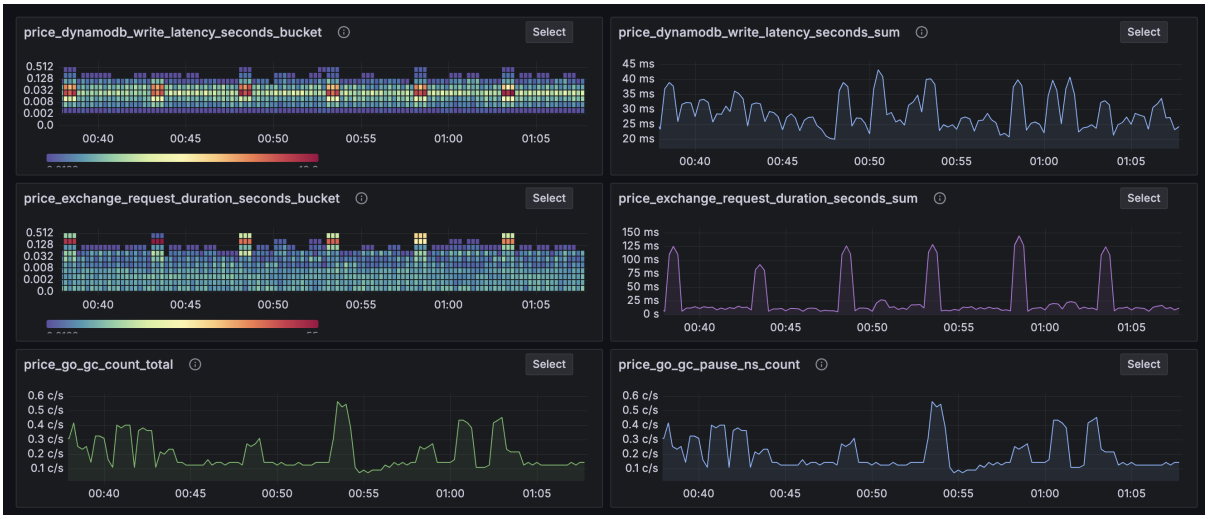
```
// Increase GC percent to reduce frequency but increase work per cycle
debug.SetGCPercent(200)  // Default is 100
```

- **Result**: More stable memory utilization and reduced GC pause times

- Before and after comparison:



VS



5.4 Concurrent Processing Refinement

Further analysis of the concurrent processing model revealed potential improvements:

- **Problem**: Direct HTTP response handling in goroutines
- **Solution**:
  - Refined the concurrent processing model
  - Added cache warmup functionality for frequently accessed assets
- **Result**: More stable response times and reduced error rates
  - Before and after comparison:

Label	# Samp...	Average	Median	90% Li...	95% Li...	99% Li...	Min	Maxim...	Error %	Throug...	Receive...	Sent K...
GET H...	10000	29	24	49	67	92	14	150	0.00%	1645.8...	415.28	280.54
GET M...	6400	32	25	61	75	95	15	127	0.00%	1554.9...	397.98	266.57
GET C...	2400	35	27	68	78	98	15	124	0.00%	870.8/...	220.17	149.67
TOTAL	18800	31	25	56	73	95	14	150	0.00%	3094.1...	784.75	529.02

vs

Label	# Samp...	Average	Median	90% Li...	95% Li...	99% Li...	Min	Maxim...	Error %	Throug...	Receive...	Sent K...
GET H...	10000	23	21	35	41	49	14	68	0.00%	1614.2...	407.31	273.58
GET M...	6400	23	21	38	42	51	14	78	0.00%	1556.8...	397.54	265.38
GET C...	2400	24	21	40	44	52	15	66	0.00%	892.9/...	225.74	152.59
TOTAL	18800	23	21	37	42	51	14	78	0.00%	3034.7...	769.06	515.89

6. Performance Test Results

Testing progression showed the system's scaling characteristics under various loads:

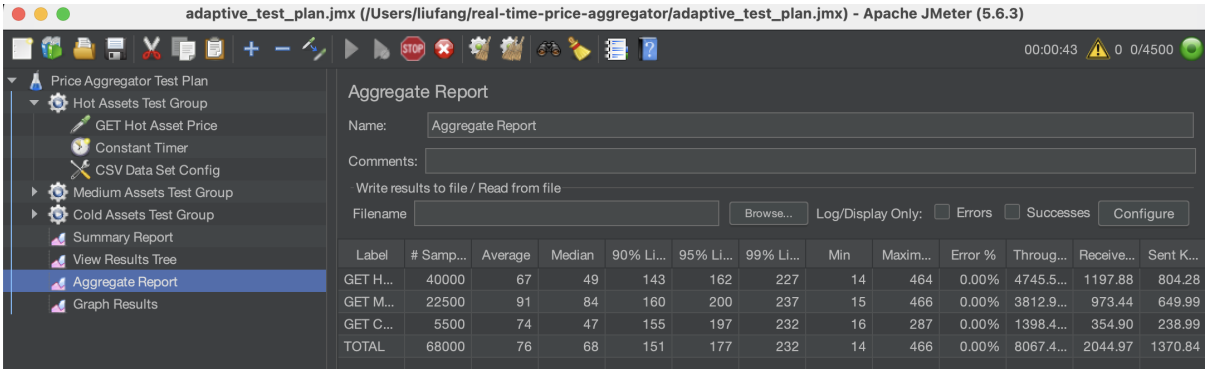
1. **Initial Tests**: Balanced performance with modest throughput (~3034 requests/sec total)

Label	# Samp...	Average	Median	90% Li...	95% Li...	99% Li...	Min	Maxim...	Error %	Throug...	Receive...	Sent K...
GET H...	10000	23	21	35	41	49	14	68	0.00%	1614.2...	407.31	273.58
GET M...	6400	23	21	38	42	51	14	78	0.00%	1556.8...	397.54	265.38
GET C...	2400	24	21	40	44	52	15	66	0.00%	892.9/...	225.74	152.59
TOTAL	18800	23	21	37	42	51	14	78	0.00%	3034.7...	769.06	515.89

2. **Increased Load**: System scaled to handle more requests (~6149 requests/sec) with proportional latency increase

Label	# Samp...	Average	Median	90% Li...	95% Li...	99% Li...	Min	Maxim...	Error %	Throug...	Receive...	Sent K...
GET H...	22500	29	25	49	56	74	14	169	0.00%	3556.7...	896.75	602.81
GET M...	10000	34	29	54	65	83	14	171	0.00%	2368.5...	602.79	403.76
GET C...	6400	36	31	57	67	88	15	176	0.00%	1881.8...	475.77	321.60
TOTAL	38900	31	27	52	61	78	14	176	0.00%	6149.2...	1554.84	1045.17

3. **Heavy Load**: Throughput continued to increase (~8067 requests/sec) with higher latency
   - **Cold Tier Limitation**: Discovered a scaling limit specific to cold-tier assets when reaching ~5500 concurrent requests



The most significant finding was the cold-tier scaling limitation. When testing with high concurrency (1000 requests × 10 threads) for cold assets, the system became unstable. Reducing the load allowed the system to resume normal operation.

7. Root Cause Analysis

7.1 Cache Hit Rate Issues

- **Problem**: Inefficient cache strategy causing high miss rates
- **Root Cause**: Mismatch between refresh intervals and cache TTL values
- **Solution**: Aligned cache TTL with refresh intervals for each tier

7.2 DynamoDB Access Patterns

- **Problem**: Suboptimal DynamoDB access during peak loads
- **Root Cause**: Individual gets instead of batch operations
- **Solution**: Implemented batch get methods and cache prewarming for hot assets

8. System Behavior Analysis

Analysis of monitoring metrics revealed several important patterns:

1. **Resource Exhaustion**: Spikes in goroutines and memory usage correlate with test phases, suggesting resource limitations when handling many cold asset requests simultaneously
2. **DynamoDB Latency**: Periodic spikes in DynamoDB write latency indicating possible throttling when processing numerous cold assets
3. **Circuit Breaker Behavior**: Circuit breakers didn't trip during tests, suggesting bottlenecks elsewhere in the system

9. Distributed System Features

The implementation includes several key distributed system patterns:

1. **Circuit Breaker Pattern**: Enhances fault tolerance by preventing cascading failures when external APIs experience issues
2. **Microservices Approach**: Separation of concerns with dedicated components
3. **CQRS Pattern**: Separation of read and write operations for optimized performance
4. **Tiered Caching Strategy**: Multi-level caching aligned with access patterns

10. System Trade-offs

Several architectural trade-offs were considered during development:

1. **Lambda vs. EC2 for Automatic Refreshes**:
    - Considered using AWS Lambda for POST/refresh operations
    - Rejected due to CloudWatch's minimum interval of 1 minute (incompatible with 5s/30s refresh requirements) and cost considerations
    - Implemented using Go tickers on EC2 instances instead
2. **Auto Scaling Group Implementation**:
    - Considered implementing ASG for dynamic scaling
    - Postponed due to system not yet reaching consistent bottlenecks and implementation complexity
3. **Kafka Integration**:
    - Considered using Kafka for decoupling components
    - Rejected due to potential data freshness issues when users request real-time prices
    - The Redis/DynamoDB solution proved sufficient for current scale
4. **DynamoDB Selection**:
    - Chose DynamoDB for its scalability, managed service benefits, and compatibility with the application's access patterns
5. **Throughput vs. Latency**:
    - Current system configuration balances throughput and latency
    - Primary optimization target is response time, with throughput as secondary consideration

11. Future Work

Several improvements are planned for future iterations:

1. **Exchange API Integration**: Create more mock exchange servers or integrate with real APIs to further test system behavior
2. **Scale Testing**: Increase symbol count to 10,000+ to simulate large-scale exchanges
3. **Kafka Integration**: Re-evaluate Kafka implementation to address the data freshness challenges
4. **Auto Scaling Implementation**: Implement and test auto-scaling capabilities for dynamic load management
5. **Performance Profiling**: Conduct deeper analysis of cold-tier asset handling to address the scaling limitation discovered
6. **Combined Load Tests**: Assessing system behavior under mixed access patterns

12. Conclusion

The current Real-Time Price Aggregator system successfully meets most performance objectives, with throughput exceeding 8,000 requests per second under optimal conditions and P95 latency remaining under target thresholds for hot and medium assets.

The primary system bottleneck appears to be in cold-tier asset handling under high concurrent load. Future optimizations will focus on addressing this limitation while maintaining the balance between throughput and latency that best serves user experience requirements.