Product Requirements Document: PortfolioPulse

Version: 1.0
Date: November 2025
Owner: Product Management
Status: Draft for Engineering Review

---

1. Executive Summary

PortfolioPulse is an AI-powered investment tracking platform that transforms how casual investors monitor and understand their portfolios. Unlike existing solutions that either charge monthly fees or provide basic tracking without insights, PortfolioPulse delivers enterprise-grade analytics using exclusively free-tier APIs and open-source AI models.

Value Proposition: Real-time portfolio monitoring with AI-driven sentiment analysis and risk scoring, accessible at zero cost while demonstrating production-quality full-stack architecture.

Key Differentiators:

- AI sentiment analysis of news impacting holdings (no competitor offers this free)
- Hybrid REST/GraphQL architecture showcasing modern API design
- Python microservices integration demonstrating polyglot architecture
- Automated risk scoring based on portfolio composition and market volatility

Business Objectives:

- Primary: Create a fully functional investment tool for 100+ beta users
- Secondary: Serve as a technical portfolio piece demonstrating: microservices architecture, AI/ML integration, real-time data processing, and scalable system design

---

2. Problem Statement

Current Pain Points:

Casual investors managing $10K-$100K portfolios face a fragmented experience. Free tools like Google Finance provide basic tracking but lack contextual intelligence—they don't answer "Why did my portfolio drop 3%?" or "Should I be concerned about this news?" Paid platforms (Seeking Alpha Premium, Morningstar) cost $200-$400 annually, targeting professional investors with features casual users don't need.

Existing free solutions fail in three areas:

1. **Context Deficit:** Price changes without news correlation or sentiment analysis
2. **Reactive Alerts:** Notifications arrive after significant movements, too late for decision-making
3. **Technical Debt:** Outdated UIs built on monolithic architectures, not leveraging modern AI capabilities

Opportunity:

The intersection of free financial APIs, open-source sentiment analysis models, and modern web frameworks enables a new category: intelligent portfolio tracking at zero cost. With 63% of Americans owning stocks and 78% of millennials checking investments on mobile devices weekly, there's demand for a solution that explains portfolio movements, not just reports them.

Jobs-to-Be-Done:

- When my portfolio value changes significantly, I need to understand why (market-wide vs. holding-specific)
- When news breaks about my holdings, I need to assess sentiment impact before making decisions
- When evaluating portfolio health, I need risk metrics beyond simple returns

---

3. User Personas & Use Cases

Persona 1: Sarah Chen - The Informed Casual Investor

Background:

- Age: 32, Product Designer at a SaaS company
- Investment Experience: 4 years, self-taught through Reddit/YouTube
- Portfolio: $45K across 12 stocks + 2 crypto holdings
- Behavior: Checks portfolio 3x weekly, reads headlines but struggles to assess impact

Goals:

- Understand portfolio performance without spending hours researching
- Receive timely alerts on significant news affecting holdings
- Visualize risk exposure across asset classes

Pain Points:

- "I see Tesla dropped 5%, but was it just them or the whole EV sector?"
- "Too many finance apps send spammy notifications for tiny movements"
- "I don't know if my portfolio is too concentrated in tech"

Success Scenario: Sarah opens PortfolioPulse on Tuesday morning, sees her portfolio is down 2.1%, and immediately sees an AI-analyzed summary: "Market-wide downturn in tech sector (-3.2%) partially offset by energy holdings (+1.8%). Negative sentiment detected in semiconductor news affecting NVDA position."

Persona 2: Marcus Rodriguez - The Developer Job Seeker

Background:

- Age: 28, Full-stack developer with 3 years experience
- Investment Experience: 2 years, small portfolio ($8K)
- Portfolio: 6 index funds + 3 individual stocks
- Primary Goal: Showcase technical skills to hiring managers

Goals:

- Demonstrate full-stack capabilities with production-quality code
- Show understanding of microservices and AI/ML integration
- Create talking points for technical interviews

Pain Points:

- "My GitHub projects look like tutorials, not real products"
- "I need to show I can architect scalable systems, not just code features"
- "Most portfolio projects don't demonstrate AI/ML integration"

Success Scenario: During an interview, Marcus walks through PortfolioPulse's architecture, explaining the REST vs. GraphQL decision, Python microservice integration, and MongoDB schema design. The interviewer is impressed by the hybrid API approach and sentiment analysis pipeline.

Primary User Scenarios

Scenario 1: Morning Portfolio Check

1. User logs in at 8:00 AM
2. Dashboard displays overnight changes with color-coded sentiment indicators
3. AI summary highlights: "2 holdings affected by negative earnings news"
4. User clicks through to see detailed sentiment analysis and news sources
5. User sets price alert for specific holding

Scenario 2: News Impact Assessment

1. User receives email alert: "Significant news detected for AAPL"
2. Opens app to see sentiment score: -0.65 (negative)
3. Reviews aggregated headlines with sentiment breakdown
4. Compares AAPL movement vs. tech sector index
5. Decides to hold position based on analysis

Scenario 3: Portfolio Risk Review

1. User navigates to Risk Dashboard
2. Views concentration risk: 45% in technology sector
3. Sees volatility score: 7.2/10 (high)
4. Reviews AI recommendations for diversification
5. Exports report for tax planning

Scenario 4: Adding New Investment

1. User searches for stock ticker (e.g., "TSLA")
2. Views current price, 52-week range, recent sentiment
3. Enters purchase details (shares, price, date)
4. System recalculates portfolio metrics and risk score
5. Confirmation screen shows updated allocation pie chart

Scenario 5: Technical Interview Walkthrough (Marcus)

1. Opens architecture documentation
2. Explains REST endpoints for CRUD operations
3. Demonstrates GraphQL for complex portfolio queries
4. Shows Python sentiment analysis logs
5. Discusses MongoDB indexing strategy for performance

---

4. Feature Requirements

F1: Multi-Asset Portfolio Tracking (P0)

User Story:
As a casual investor, I want to track stocks, ETFs, and cryptocurrencies in one unified portfolio, so that I can see my complete investment picture without switching between apps.

Acceptance Criteria:

- User can add holdings by ticker symbol with purchase price, quantity, and date
- System supports 500+ US stocks, 50+ major ETFs, and top 20 cryptocurrencies
- Real-time price updates occur every 5 minutes during market hours
- Portfolio value displays total worth, daily change ($/%), and total gain/loss
- Users can manually edit/delete transactions with audit trail
- Supports fractional shares and crypto decimals (up to 8 places)

Technical Considerations:

- **Data Sources:** Yahoo Finance API (stocks/ETFs), CoinGecko API (crypto)
- **Rate Limits:** Yahoo Finance: 2,000 calls/day; CoinGecko: 50 calls/minute
- **Caching Strategy:** Cache price data for 5 minutes, refresh on user request with rate limit check
- **Schema Design:** Separate collections for `portfolios`, `holdings`, `transactions` with references

Priority: P0 - Core functionality, must ship in Sprint 1

F2: AI-Powered Sentiment Analysis (P0)

User Story:
As an investor, I want to see sentiment analysis of news affecting my holdings, so that I can quickly assess whether market movements warrant action.

Acceptance Criteria:

- System aggregates news articles for each holding from past 24 hours
- Each article receives a sentiment score (-1.0 to +1.0) using NLP model
- Dashboard displays aggregate sentiment per holding with visual indicator (red/yellow/green)
- Users can drill down to see individual articles with sentiment scores
- Sentiment updates every 4 hours or on-demand
- Model accuracy target: 70%+ on financial news validation set

Technical Considerations:

- **Model:** Fine-tuned FinBERT (Hugging Face) for financial sentiment
- **Python Service:** FastAPI microservice exposed via REST endpoint
- **Communication:** Node.js calls Python service, caches results in Redis (2-hour TTL)
- **Training Data:** Use Financial PhraseBank dataset + labeled Yahoo Finance articles
- **Fallback:** If Python service unavailable, display "Sentiment analysis temporarily unavailable"

Priority: P0 - Key differentiator, must ship in Sprint 3

F3: Automated Risk Scoring (P1)

User Story:
As an investor concerned about portfolio volatility, I want an automated risk score, so that I can understand my exposure without manual calculations.

Acceptance Criteria:

- System calculates portfolio risk score (1-10 scale) based on:
  - Asset volatility (historical 30-day standard deviation)
  - Concentration risk (Herfindahl index for diversification)

    - Sector exposure (tech > 40% increases score)
- Risk score updates daily at market close
- Dashboard displays score with benchmark comparison (S&P 500 = 5.0 baseline)
- Detailed breakdown shows risk contribution per holding
- Users can simulate risk impact by adding/removing hypothetical holdings

Technical Considerations:

- **Algorithm:** Weighted formula: 40% volatility + 30% concentration + 30% sector exposure
- **Data Requirements:** 30-day historical price data (Yahoo Finance)
- **Computation:** Run as scheduled job (Node-Cron) at 6:00 PM ET daily
- **Storage:** Store risk scores in `riskMetrics` collection with timestamp

Priority: P1 - Important for portfolio health, Sprint 3

F4: Email Alert System (P1)

User Story:
As a busy professional, I want email alerts for significant portfolio events, so that I don't need to constantly check the app.

Acceptance Criteria:

- Users can configure alert thresholds:
    - Portfolio total change (default: ±3%)
    - Individual holding change (default: ±5%)
    - High-impact news sentiment (score < -0.7 or > +0.7)
- Maximum 3 emails per day to prevent spam
- Email includes summary data + direct link to dashboard
- Users can disable alerts or adjust thresholds in settings
- Alerts sent within 15 minutes of trigger event

Technical Considerations:

- **Email Service:** SendGrid (free tier: 100 emails/day)
- **Trigger Logic:** Background job checks conditions every 15 minutes
- **Template:** HTML email with branded header, data table, CTA button
- **Unsubscribe:** One-click unsubscribe link compliant with CAN-SPAM

Priority: P1 - Engagement driver, Sprint 4

F5: Performance Analytics Dashboard (P1)

User Story:
As an investor tracking long-term performance, I want visualizations of portfolio growth, so that I can evaluate my investment strategy.

Acceptance Criteria:

- Line chart showing portfolio value over time (1W, 1M, 3M, 1Y, All views)
- Pie chart displaying current asset allocation by ticker and sector
- Bar chart comparing individual holding performance (% gain/loss)
- Summary metrics: Total return ($/%), annualized return, best/worst performers
- Ability to export data as CSV for tax purposes
- Charts render in <2 seconds with 1 year of daily data points

Technical Considerations:

- **Charting Library:** Recharts (React-optimized, 45KB gzipped)
- **Data Aggregation:** GraphQL query fetches historical snapshots from `portfolioHistory` collection
- **Performance:** Implement pagination for >365 data points, lazy-load charts on scroll
- **Export:** Generate CSV server-side to avoid browser memory issues

Priority: P1 - Portfolio showcase feature, Sprint 4

F6: User Authentication (P0)

User Story:
As a user, I want secure login with my email, so that my portfolio data remains private.

Acceptance Criteria:

- Email/password registration with validation (min 8 chars, 1 number, 1 special char)
- JWT-based authentication with 7-day expiry
- Password reset via email with 1-hour expiry token
- Session management with automatic logout after 30 days inactivity
- OAuth support for Google login (stretch goal for Sprint 2)

Technical Considerations:

- **Library:** Passport.js with Local and JWT strategies
- **Password Storage:** Bcrypt with salt rounds = 12
- **Token Management:** Store refresh tokens in MongoDB with user reference
- **Security Headers:** Helmet.js for XSS/CSRF protection

Priority: P0 - Foundation for all features, Sprint 1

---

5. Technical Architecture

System Architecture Overview

Component Diagram (Description):

```
[React Frontend] <--> [API Gateway (Express)] <--> [MongoDB]
                              |
                              ├─> [Python Sentiment Service]
                              ├─> [REST Controllers]
                              └─> [GraphQL Resolver]
                                   |
                                   └─> [External APIs (Yahoo, NewsAPI)]
```

Architecture Pattern: Hybrid monolith + microservices

- **Monolith Core:** Node.js/Express handles authentication, portfolio CRUD, external API orchestration
- **Microservice:** Python FastAPI service for CPU-intensive sentiment analysis (decoupled for independent scaling)

REST vs. GraphQL Decision Rationale

REST Endpoints (70% of API surface):

Use Cases:

- Simple CRUD operations (create holding, update transaction, delete portfolio)
- Authentication flows (login, register, password reset)
- Webhook receivers (external API callbacks)
- File uploads (CSV import for bulk transactions)

Rationale:

- **Caching:** Standard HTTP caching headers work out-of-the-box
- **Simplicity:** Clear URL structure (`POST /api/portfolios`, `GET /api/holdings/:id`)
- **Tool Support:** Better Postman/Swagger documentation, easier for other developers to understand
- **Performance:** No query parsing overhead for straightforward operations

Example REST Endpoints:

```
POST   /api/auth/register
POST   /api/auth/login
GET    /api/portfolios/:userId
POST   /api/holdings
PUT    /api/holdings/:id
DELETE /api/holdings/:id
GET    /api/news/:ticker
```

GraphQL Endpoint (30% of API surface):

Use Cases:

- Complex nested queries (portfolio + holdings + transactions + current prices + sentiment in one request)
- Dashboard data aggregation (multiple data sources combined)
- Flexible client requirements (mobile app may need subset of fields)
- Real-time subscriptions (price updates via WebSocket)

Rationale:

- **Over-fetching Prevention:** Dashboard needs 12 data points from 4 collections; REST would require 4 sequential calls or one bloated endpoint
- **Developer Experience:** Frontend can request exactly what it needs: `{ portfolio { holdings { ticker, quantity, currentPrice, sentiment } } }`
- **Evolution:** Easy to add fields without versioning (`holdings { ticker, quantity, NEW_FIELD }`)
- **Showcase Value:** Demonstrates understanding of modern API design patterns for portfolio project

Example GraphQL Query:

graphql

```graphql
query DashboardData($userId: ID!) {
  portfolio(userId: $userId) {
    totalValue
    dailyChange
    holdings {
      ticker
      quantity
      purchasePrice
      currentPrice
      sentiment {
        score
        articles { title, sentiment }
      }
    }
    riskMetrics {
      overallScore
      volatility
    }
  }
}
```

Hybrid Decision Matrix:

Operation Type	API Choice	Reason
User registration	REST	Standard form submission, caching
Add holding	REST	Simple create, no nested data
Dashboard load	GraphQL	6+ data sources, prevent over-fetching
Holdings list	GraphQL	Client needs flexible field selection
Delete transaction	REST	Single operation, HTTP verb clarity
Search holdings	GraphQL	Filtering + sorting + pagination

Python Microservice Integration

Communication Protocol:

- **Inter-Service:** HTTP REST (Node → Python FastAPI at `http://localhost:8000`)
- **Message Format:** JSON with schema validation via Pydantic models
- **Error Handling:** Python service returns standard HTTP status codes; Node retries 3x with exponential backoff

Request Flow:

1. Node.js receives client request for sentiment analysis
2. Checks Redis cache for recent sentiment (TTL: 4 hours)
3. If cache miss, calls Python service: `POST /analyze` with article text
4. Python service loads FinBERT model, computes sentiment, returns score
5. Node.js caches result and returns to client

Deployment:

- **Development:** Both services run locally (Node on port 3000, Python on port 8000)
- **Production:** Deploy Python service as separate container, Node service calls via internal DNS

Authentication & Authorization

Approach: JWT-based stateless authentication

- **Login Flow:** User submits credentials → Node verifies password → Issues JWT with 7-day expiry + refresh token
- **Authorization:** Middleware extracts JWT from `Authorization: Bearer <token>` header, verifies signature, attaches `req.user`
- **Protected Routes:** All `/api/*` routes except `/auth/login` and `/auth/register` require valid JWT
- **Refresh Mechanism:** Refresh token (30-day expiry) stored in httpOnly cookie, used to issue new JWT

Security Measures:

- Passwords hashed with Bcrypt (salt rounds: 12)
- JWT secret stored in environment variable (256-bit random string)
- Rate limiting: 5 failed login attempts → 15-minute lockout
- HTTPS enforced in production (Helmet.js middleware)

---

6. AI/ML Implementation Details

Sentiment Analysis Model

Model Selection: FinBERT (Financial BERT)

- **Rationale:** Pre-trained on financial news corpus, outperforms generic BERT on finance-specific language (e.g., "bearish", "overvalued")
- **Source:** Hugging Face `ProsusAI/finbert` (open-source, Apache 2.0 license)
- **Model Size:** 438MB (requires 2GB RAM for inference)

Training Approach:

- **Phase 1 (Week 3):** Use pre-trained FinBERT weights without fine-tuning (baseline accuracy ~65%)
- **Phase 2 (Post-MVP):** Fine-tune on custom dataset:
  - Collect 1,000 Yahoo Finance articles for top 50 holdings
  - Label sentiment manually: positive/neutral/negative
  - Fine-tune for 3 epochs using Hugging Face Trainer API
  - Target: 75%+ accuracy on validation set

Inference Pipeline:


python

```python
# Python FastAPI endpoint
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import torch

model = AutoModelForSequenceClassification.from_pretrained("ProsusAI/finbert")
tokenizer = AutoTokenizer.from_pretrained("ProsusAI/finbert")

@app.post("/analyze")
def analyze_sentiment(text: str):
    inputs = tokenizer(text, return_tensors="pt", truncation=True, max_length=512)
    outputs = model(**inputs)
    scores = torch.nn.functional.softmax(outputs.logits, dim=-1)
    sentiment = scores.argmax().item()  # 0=negative, 1=neutral, 2=positive
    confidence = scores.max().item()
    return {"sentiment": (sentiment - 1) / 1.0, "confidence": confidence}
```

Risk Scoring Algorithm

Methodology: Composite score from three components

Formula:

```
Risk Score = (0.4 × Volatility Score) + (0.3 × Concentration Score) + (0.3 × Sector Exposure Score)
```

Component Calculations:

1. **Volatility Score (1-10):**
   - Calculate 30-day standard deviation of daily returns per holding
   - Weight by portfolio allocation
   - Normalize: $\sigma < 10\% \rightarrow 1$, $\sigma > 40\% \rightarrow 10$
2. **Concentration Score (1-10):**
   - Herfindahl Index: $HHI = \Sigma(\text{weight}^2)$ for each holding
   - $HHI < 0.15$ (diversified) $\rightarrow 1$, $HHI > 0.40$ (concentrated) $\rightarrow 10$
3. **Sector Exposure Score (1-10):**
   - Map each ticker to GICS sector (via Yahoo Finance API)
   - Max sector weight $< 25\% \rightarrow 1$, $> 60\% \rightarrow 10$

Example Calculation:

- Portfolio: 60% AAPL, 20% MSFT, 20% GOOGL (all tech)
- Volatility: 25% avg → Score: 6.3
- Concentration: HHI = 0.44 → Score: 10
- Sector: 100% tech → Score: 10

- **Final Risk Score: (0.4×6.3) + (0.3×10) + (0.3×10) = 8.5/10 (High Risk)**

Python-Node.js Communication

Protocol: Synchronous HTTP with async/await on Node side

Node.js Client:


javascript

```javascript
const axios = require('axios');

async function getSentiment(articleText) {
  try {
    const response = await axios.post('http://localhost:8000/analyze', {
      text: articleText
    }, { timeout: 5000 });
    return response.data.sentiment;
  } catch (error) {
    console.error('Sentiment service error:', error.message);
    return 0; // Neutral fallback
  }
}
```

Error Handling Strategy:

- **Timeout:** 5-second limit per request
- **Retry Logic:** 3 attempts with 1s, 2s, 4s delays (exponential backoff)
- **Circuit Breaker:** After 10 consecutive failures, stop calling Python service for 5 minutes
- **Fallback:** Return neutral sentiment (0.0) and log error for monitoring

---

Model Update & Versioning

Versioning Strategy:

- **Model Versions:** Semantic versioning (v1.0.0, v1.1.0)
- **Endpoint Versioning:** `/api/v1/analyze` allows future breaking changes
- **Deployment:** Blue-green deployment for Python service (run v1 and v2 simultaneously, gradually shift traffic)

Update Process (Post-MVP):

1. Collect new labeled data monthly
2. Retrain model offline
3. Evaluate on holdout set (min 70% accuracy threshold)
4. Deploy as new version with traffic split (10% v2, 90% v1)

5. Monitor error rates and user feedback
6. Full cutover after 1 week of stable performance

---

7. Data Architecture

MongoDB Schema Design

Collections:

1. users

javascript

```
{
  _id: ObjectId,
  email: String (unique, indexed),
  passwordHash: String,
  createdAt: Date,
  lastLogin: Date,
  preferences: {
    alertThreshold: Number,
    emailEnabled: Boolean
  }
}
```

2. portfolios

javascript

```
{
  _id: ObjectId,
  userId: ObjectId (indexed, ref: users),
  name: String,
  createdAt: Date,
  totalValue: Number, // Cached for performance
  dailyChange: Number,
  lastUpdated: Date
}
```

3. holdings

javascript

```
{
  _id: ObjectId,
  portfolioId: ObjectId (indexed, ref: portfolios),
  ticker: String (indexed),
  assetType: String, // 'stock' | 'crypto' | 'etf'
  quantity: Number,
  averageCost: Number, // Calculated from transactions
  currentPrice: Number, // Cached from API
  lastPriceUpdate: Date
}
```

4. transactions

javascript

```
{
  _id: ObjectId,
  holdingId: ObjectId (indexed, ref: holdings),
  type: String, // 'buy' | 'sell'
  quantity: Number,
  pricePerUnit: Number,
  date: Date (indexed),
  createdAt: Date
}
```

5. sentimentData

javascript

```
{
  _id: ObjectId,
  ticker: String (indexed),
  sentimentScore: Number, // -1.0 to +1.0
  articles: [{
    title: String,
    url: String,
    sentiment: Number,
    publishedAt: Date
  }],
  calculatedAt: Date,
  expiresAt: Date // TTL index for auto-deletion after 24 hours
}
```

6. riskMetrics

javascript

```
{
  _id: ObjectId,
  portfolioId: ObjectId (indexed),
  overallScore: Number, // 1-10
  components: {
    volatility: Number,
    concentration: Number,
    sectorExposure: Number
  },
  calculatedAt: Date (indexed)
}
```

Indexing Strategy

Critical Indexes:

- `users.email` (unique) - Fast login lookups
- `portfolios.userId` - User's portfolio queries
- `holdings.portfolioId` - Holdings for specific portfolio
- `holdings.ticker` - Price updates for specific tickers
- `transactions.holdingId + transactions.date` (compound) - Transaction history queries
- `sentimentData.ticker + sentimentData.calculatedAt` (compound) - Recent sentiment lookup
- `sentimentData.expiresAt` (TTL) - Auto-delete stale sentiment data

Query Patterns:

- Most frequent: `holdings.find({ portfolioId: X })` - optimized by index
- Dashboard load: Aggregation pipeline joins portfolios + holdings + sentiment (uses indexes)
- Risk calculation: Batch query all holdings for portfolio (indexed by portfolioId)

API Data Caching

Challenge: Free-tier rate limits

- Yahoo Finance: 2,000 calls/day (≈1 call every 43 seconds)
- NewsAPI: 100 requests/day
- CoinGecko: 50 calls/minute

Caching Strategy:

Layer 1: Redis (In-Memory)

- **Price Data:** TTL = 5 minutes (during market hours), 1 hour (after hours)
- **News Data:** TTL = 4 hours (sentiment doesn't change rapidly)
- **Key Pattern:** `price:{ticker}:{date}`, `sentiment:{ticker}:{timestamp}`

Layer 2: MongoDB (Persistent)

- **Historical Prices:** Store daily closing prices in `priceHistory` collection
- **Sentiment History:** `sentimentData` with 24-hour retention (TTL index)
- **Use Case:** Backfill data for charts without re-fetching from APIs

Rate Limit Management:

- **Request Queue:** FIFO queue for API calls, processes max 1 request every 45 seconds for Yahoo Finance
- **Priority System:** User-initiated requests (dashboard load) > background jobs (daily risk calc)
- **Fallback:** If rate limit hit, serve stale cache data with "Last updated 30 minutes ago" disclaimer

---

Data Refresh Frequencies

Data Type	Refresh Frequency	Trigger
Stock prices (market hours)	5 minutes	Scheduled job + user request
Stock prices (after hours)	1 hour	Scheduled job
Crypto prices	10 minutes	Scheduled job (more volatile)
News articles	4 hours	Scheduled job
Sentiment scores	4 hours	After news refresh
Risk scores	Daily (6 PM ET)	Scheduled job
Portfolio totals	On price update	Calculated field

---

8. API Integration Strategy

External APIs & Limitations

1. Yahoo Finance API (via `yahoo-finance2` npm package)

- **Use Case:** Stock/ETF prices, historical data, company info

- **Rate Limit:** 2,000 requests/day (approximately 1.4 requests/minute)
- **Limitations:** No official API (uses web scraping), occasional schema changes
- **Data Latency:** 15-minute delay for real-time quotes on free tier
- **Cost:** Free

2. CoinGecko API

- **Use Case:** Cryptocurrency prices (BTC, ETH, etc.)
- **Rate Limit:** 50 calls/minute (generous for our use case)
- **Limitations:** Pro features (historical minute-level data) not available on free tier
- **Data Latency:** Real-time for top 100 coins
- **Cost:** Free

3. NewsAPI.org

- **Use Case:** Financial news articles by keyword (ticker symbol)
- **Rate Limit:** 100 requests/day (tight constraint)
- **Limitations:** 1-month historical data max, no sentiment scoring
- **Data Latency:** Near real-time (5-minute delay)
- **Cost:** Free (Developer plan)

4. Alpha Vantage (Backup)

- **Use Case:** Historical stock data if Yahoo Finance fails
- **Rate Limit:** 25 requests/day (very restrictive)
- **Limitations:** 5 API calls per minute
- **Data Latency:** End-of-day data
- **Cost:** Free

Rate Limit Management

Implementation:

1. Request Queue System

javascript

```
class RateLimitedAPIClient {
  constructor(maxRequestsPerMinute) {
    this.queue = [];
    this.requestsInWindow = 0;
    this.windowStart = Date.now();
    this.maxRequests = maxRequestsPerMinute;
  }

  async request(apiCall) {
    // Check if we're within rate limit
    if (this.requestsInWindow >= this.maxRequests) {
      const waitTime = 60000 - (Date.now() - this.windowStart);
      await this.sleep(waitTime);
      this.resetWindow();
    }

    this.requestsInWindow++;
    return await apiCall();
  }

  resetWindow() {
    this.windowStart = Date.now();
    this.requestsInWindow = 0;
  }
}
```

2. Request Batching

- **Strategy:** Group multiple ticker price requests into single API call where possible
- **Example:** Yahoo Finance supports batch quotes: `yf.quote(['AAPL', 'MSFT', 'GOOGL'])`
- **Benefit:** Reduce 3 API calls to 1

3. Intelligent Scheduling

- **Market Hours (9:30 AM - 4 PM ET):** Refresh prices every 5 minutes (max 78 calls/day per ticker)
- **After Hours:** Reduce to 1-hour intervals (saves 90% of quota)
- **User-Triggered:** Bypass cache only if last update >5 minutes ago

---

Fallback Strategies

Scenario 1: Yahoo Finance Unavailable

- **Detection:** HTTP 5xx errors or timeout after 10 seconds
- **Fallback:** Switch to Alpha Vantage API (max 25 tickers/day)
- **User Communication:** Display banner: "Using backup data source, prices may be delayed"

- **Recovery:** Retry Yahoo Finance every 15 minutes

Scenario 2: NewsAPI Daily Limit Exceeded

- **Detection:** 429 status code or daily quota message
- **Fallback:** Display cached news from last 24 hours
- **User Communication:** "News updates paused until tomorrow (rate limit reached)"
- **Mitigation:** Implement priority system - only fetch news for holdings user views, not all portfolio holdings proactively

Scenario 3: Rate Limit Hit During Dashboard Load

- **Detection:** Queue wait time exceeds 30 seconds
- **Fallback:** Return cached data with timestamp: "Prices as of 20 minutes ago"
- **User Action:** "Refresh now" button manually triggers cache bypass
- **Prevention:** Pre-cache top 100 stocks every 5 minutes (background job)

Scenario 4: API Schema Change (Yahoo Finance)

- **Detection:** JSON parsing errors or missing fields
- **Fallback:** Log error, return null, display "Price unavailable"
- **Alert:** Slack notification to developers (use Slack webhook)
- **Resolution:** Emergency patch to update parsing logic

Data Normalization Layer

Purpose: Abstract external API differences into consistent internal format

Normalized Price Response:


javascript

```
{
  ticker: "AAPL",
  price: 175.43,
  change: 2.15,
  changePercent: 1.24,
  volume: 52431200,
  timestamp: "2025-11-13T16:00:00Z",
  source: "yahoo" // or "coingecko", "alphavantage"
}
```

Adapter Pattern Implementation:


javascript

```
class PriceDataAdapter {
  static normalize(rawData, source) {
    switch(source) {
      case 'yahoo':
        return {
          ticker: rawData.symbol,
          price: rawData.regularMarketPrice,
          change: rawData.regularMarketChange,
          changePercent: rawData.regularMarketChangePercent,
          volume: rawData.regularMarketVolume,
          timestamp: new Date(rawData.regularMarketTime * 1000).toISOString(),
          source: 'yahoo'
        };
      case 'coingecko':
        return {
          ticker: rawData.symbol.toUpperCase(),
          price: rawData.current_price,
          change: rawData.price_change_24h,
          changePercent: rawData.price_change_percentage_24h,
          volume: rawData.total_volume,
          timestamp: new Date().toISOString(),
          source: 'coingecko'
        };
      // Add alphavantage case
    }
  }
}
```

Benefits:

- Frontend code doesn't know/care which API provided data
- Easy to swap APIs without changing client code
- Consistent error handling across all data sources

---

9. Non-Functional Requirements

Performance Targets

Response Time Requirements:

Endpoint/Action	Target	Maximum Acceptable
Initial page load (homepage)	<2 seconds	3 seconds
Dashboard data load	<1.5 seconds	2.5 seconds
Add holding (CRUD)	<500ms	1 second
GraphQL complex query	<1 second	2 seconds
Sentiment analysis (with cache hit)	<200ms	500ms
Sentiment analysis (cache miss)	<3 seconds	5 seconds
Price update background job	N/A (async)	Complete within 2 min

Optimization Strategies:

- **Frontend:** Code splitting with React.lazy(), minimize bundle size (<250KB gzipped)
- **Backend:** Database query optimization with explain() analysis, N+1 query prevention
- **Caching:** Redis for hot data, MongoDB indexes for cold data
- **CDN:** Serve static assets via Cloudflare (free tier)

Security Requirements

Authentication & Authorization:

- **Password Policy:** Minimum 8 characters, 1 uppercase, 1 number, 1 special character
- **Session Management:** JWT tokens with 7-day expiry, refresh tokens with 30-day expiry
- **Token Storage:** JWT in httpOnly cookie (prevents XSS), refresh token in secure httpOnly cookie
- **Account Lockout:** 5 failed login attempts → 15-minute lockout

Data Encryption:

- **In Transit:** HTTPS/TLS 1.3 enforced (redirects from HTTP)
- **At Rest:** MongoDB encryption at rest (Atlas default), passwords hashed with Bcrypt (salt rounds: 12)
- **API Keys:** Stored in environment variables, never committed to Git (use .env file)

Input Validation:

- **Server-Side:** Joi schema validation for all request bodies
- **Sanitization:** Escape HTML/SQL to prevent injection attacks (use validator.js)
- **Rate Limiting:** 100 requests per 15 minutes per IP (Express rate-limit middleware)

OWASP Top 10 Mitigations:

- **XSS:** Content Security Policy headers via Helmet.js
- **CSRF:** SameSite cookie attribute for JWT
- **SQL Injection:** N/A (NoSQL, but use parameterized queries for MongoDB)
- **Sensitive Data Exposure:** No passwords/tokens logged, redact from error messages

Compliance:

- **GDPR-Ready:** User data deletion endpoint (`DELETE /api/users/:id`), export user data as JSON
- **Privacy:** No third-party analytics tracking (for portfolio project, can add Google Analytics post-MVP)

Scalability Considerations

Current MVP Scope: 100-500 users

Bottleneck Analysis:

1. **External API Rate Limits (Primary Constraint)**
    - **Solution:** Aggressive caching + request queuing
    - **Scale Path:** Upgrade to paid APIs when >1,000 users (Yahoo Finance Pro: $30/month for 10K calls/day)
2. **Database Queries**
    - **Solution:** Proper indexing, avoid full collection scans
    - **Scale Path:** MongoDB sharding by userId when >100K users
3. **Python Sentiment Service**
    - **Solution:** Load model once on startup (not per request)
    - **Scale Path:** Deploy multiple Python containers behind load balancer when >500 concurrent users

Horizontal Scaling Strategy (Post-MVP):

- **Node.js:** Stateless design allows multiple instances behind Nginx
- **Python Service:** Run 3+ containers with round-robin load balancing
- **Database:** MongoDB Atlas auto-scaling (increase tier as needed)
- **Caching:** Redis Cluster for distributed caching

Monitoring Thresholds:

- Alert when API rate limit utilization >80%
- Alert when average response time >2 seconds
- Alert when database connections >80% of pool size

Browser Compatibility

Supported Browsers:

- **Desktop:** Chrome 90+, Firefox 88+, Safari 14+, Edge 90+
- **Mobile:** iOS Safari 14+, Chrome Android 90+

Polyfills Required:

- None (targeting modern browsers, ES6+ features native)

Progressive Enhancement:

- **JavaScript Disabled:** Display message "JavaScript required for PortfolioPulse"
- **Graceful Degradation:** Charts fail → Show data table fallback

---

Mobile Responsiveness

Breakpoints:

- **Desktop:** ≥1024px (full dashboard, 3-column layout)
- **Tablet:** 768-1023px (2-column layout, condensed navigation)
- **Mobile:** <768px (single column, hamburger menu, touch-optimized buttons)

Mobile-Specific Optimizations:

- **Touch Targets:** Minimum 44×44px for all interactive elements
- **Image Optimization:** Serve WebP format with fallback to PNG
- **Viewport:** `<meta name="viewport" content="width=device-width, initial-scale=1">`
- **Performance:** Lazy load below-the-fold charts, reduce API calls on mobile (only refresh on user pull-to-refresh)

Testing:

- Manual testing on iOS Safari and Chrome Android
- Chrome DevTools device emulation for additional form factors
- Lighthouse mobile performance score target: 80+

---

10. Development Roadmap

Sprint 1: Core Infrastructure (Weeks 1-2)

Goals: Foundation for all features, basic portfolio tracking

Deliverables:

- User authentication (register, login, JWT middleware)
- MongoDB schema setup with indexes
- REST API for portfolios and holdings CRUD
- React frontend skeleton with routing (React Router)
- Basic dashboard UI (no data visualization yet)

Technical Tasks:

1. **Backend (Node.js/Express):**
    - Initialize Express app with Helmet, CORS, body-parser
    - Implement Passport.js Local + JWT strategies
    - Create Mongoose models for users, portfolios, holdings, transactions
    - Build REST controllers: `AuthController`, `PortfolioController`, `HoldingController`
    - Write unit tests for auth logic (Jest)
2. **Frontend (React):**
    - Create-react-app setup with Tailwind CSS
    - Implement login/register pages with form validation
    - Build protected route wrapper (redirect if not authenticated)
    - Create basic dashboard shell with mock data
3. **DevOps:**
    - Set up MongoDB Atlas cluster (free tier M0)
    - Configure environment variables (.env file structure)
    - GitHub repo with .gitignore for node_modules, .env

Success Criteria:

- User can register, log in, and see empty dashboard
- Manual testing shows JWT tokens issued correctly
- Database has 3 collections with proper indexes
- Code coverage: 60% for backend authentication logic

Risks & Mitigation:

- **Risk:** JWT implementation bugs (e.g., token not expiring)
    - **Mitigation:** Use battle-tested library (jsonwebtoken), write expiry tests
- **Risk:** MongoDB connection issues in Atlas
    - **Mitigation:** Whitelist IP addresses, test connection string locally first

---

Sprint 2: API Integrations & Data Pipeline (Weeks 3-4)

Goals: Real-time price data, news aggregation, rate limit management

Deliverables:

- Yahoo Finance API integration for stock prices
- CoinGecko API integration for crypto prices
- NewsAPI integration for ticker-based news
- Redis caching layer for API responses
- Background job scheduler (Node-Cron) for price updates
- Data normalization layer (adapter pattern)

Technical Tasks:

1. **Backend (API Integration):**
    - Install `yahoo-finance2`, `coingecko-api`, `newsapi` npm packages
    - Build `PriceService` class with rate limit queue
    - Implement `NewsService` with caching logic
    - Create adapter functions to normalize API responses
    - Set up Redis client (use `ioredis` package)
2. **Background Jobs:**
    - Create scheduled job: Update prices every 5 minutes during market hours
    - Create scheduled job: Fetch news every 4 hours for all tracked tickers
    - Implement job queue with priority system (user requests > background jobs)
3. **Frontend:**
    - Display real-time prices on dashboard
    - Show price change indicators (green/red arrows)
    - Add manual refresh button with loading spinner
4. **Testing:**
    - Mock external API calls in tests (nock library)
    - Test rate limit queue behavior under load
    - Integration test: Add holding → Price fetched and displayed

Success Criteria:

- Dashboard shows live prices for added holdings (15-min delay acceptable)
- Redis cache hit rate >70% for price requests
- Background jobs run without blocking user requests
- No rate limit errors during normal usage (5 users testing)

Dependencies:

- Sprint 1 must be complete (authentication + basic CRUD)

Risks & Mitigation:

- **Risk:** Yahoo Finance schema changes break parsing
    - **Mitigation:** Wrap API calls in try-catch, log errors, implement Alpha Vantage fallback
- **Risk:** Free-tier Redis memory limit (25MB) exceeded
    - **Mitigation:** Set aggressive TTL (5 minutes for prices), monitor memory usage

---

Sprint 3: AI Microservice & Sentiment Analysis (Weeks 5-6)

Goals: Deploy Python sentiment analysis, integrate with Node.js backend

Deliverables:

- Python FastAPI microservice with FinBERT model
- Sentiment analysis endpoint exposed to Node.js
- Sentiment data collection in MongoDB
- Dashboard sentiment indicators (color-coded badges)
- Risk scoring algorithm implementation

Technical Tasks:

1. **Python Service:**
   - Create FastAPI app with `/analyze` POST endpoint
   - Download and load FinBERT model from Hugging Face
   - Implement batch sentiment analysis (process multiple articles in one call)
   - Add health check endpoint `/health`
   - Dockerize Python service (optional for MVP, recommended)
2. **Backend (Node.js):**
   - Create `SentimentService` to call Python API
   - Implement retry logic with exponential backoff
   - Store sentiment results in `sentimentData` collection
   - Build `RiskService` to calculate portfolio risk score
3. **Frontend:**
   - Add sentiment badges to holdings list (red/yellow/green)
   - Create sentiment detail modal showing article-level scores
   - Add risk score widget to dashboard header
   - Implement drill-down: Click risk score → View breakdown
4. **Testing:**
   - Test Python service with sample financial articles
   - Validate sentiment accuracy on 50 manually labeled articles (target: 70%+)
   - Load test: 10 concurrent sentiment requests

Success Criteria:

- Sentiment analysis completes in <3 seconds per article
- Dashboard displays aggregate sentiment for each holding
- Risk score updates daily at 6 PM ET
- Python service uptime: 99% during testing period

Dependencies:

- Sprint 2 news integration must work (provides articles for sentiment analysis)

Risks & Mitigation:

- **Risk:** Python model loading too slow (5+ seconds on first request)
  - **Mitigation:** Load model on service startup, not per request
- **Risk:** Sentiment accuracy <70% on real articles
  - **Mitigation:** Use pre-trained FinBERT (already finance-tuned), fine-tuning can wait for post-MVP
- **Risk:** Python service crashes under load
  - **Mitigation:** Implement circuit breaker pattern in Node.js, fallback to neutral sentiment

---

Sprint 4: Dashboard UI & Email Alerts (Weeks 7-8)

Goals: Polish user experience, implement alerting, finalize analytics

Deliverables:

- Performance analytics charts (Recharts visualizations)
- Pie chart for asset allocation
- Line chart for portfolio value over time
- Email alert system with SendGrid
- User settings page (alert thresholds, email preferences)
- CSV export functionality for transactions

Technical Tasks:

1. **Frontend (Data Visualization):**
    - Install Recharts library
    - Build `PerformanceChart` component (line chart with date range selector)
    - Build `AllocationChart` component (pie chart with percentages)
    - Build `HoldingsPerformance` component (bar chart showing % gains)
    - Implement responsive chart sizing for mobile
2. **Backend (Alerts):**
    - Set up SendGrid account and verify sender email
    - Create email templates (HTML + plain text versions)
    - Build `AlertService` to check alert conditions
    - Create background job: Run alert checks every 15 minutes
    - Implement daily alert limit (max 3 emails per user)
3. **Settings Page:**
    - Build React form for alert threshold configuration
    - Add toggle for email notifications (on/off)
    - Implement "Test Alert" button (sends sample email)
4. **Polish:**
    - Add loading skeletons for slow-loading components
    - Implement error boundaries to catch React crashes
    - Add empty states ("No holdings yet, click + to add")
    - Write user documentation (README with screenshots)

Success Criteria:

- All charts render in <2 seconds with 1 year of data
- Email alerts sent within 15 minutes of trigger condition
- User can configure alert thresholds and see changes reflected immediately
- CSV export downloads successfully with all transaction data
- Mobile responsive testing passes on iPhone and Android

Dependencies:

- Sprint 3 risk scoring (needed for risk-based alerts)

Risks & Mitigation:

- **Risk:** Chart performance degrades with large datasets (>1,000 data points)
    - **Mitigation:** Implement data point sampling for >365 days, lazy load charts on scroll
- **Risk:** SendGrid free tier limit (100 emails/day) insufficient for 100 users
    - **Mitigation:** Implement 3-email-per-user daily cap, prioritize high-value alerts
- **Risk:** Scope creep (requests for additional features)
    - **Mitigation:** Document as "Post-MVP" in backlog, stay focused on P0/P1 features

---

Post-Sprint Buffer (Optional Week 9)

Purpose: Bug fixes, testing, deployment prep

Activities:

- Fix bugs discovered during Sprint 4 testing
- Write deployment documentation (environment variables, database setup)
- Create demo video for portfolio showcase (3-minute walkthrough)
- Set up production environment (Heroku/Railway/Vercel)
- Penetration testing for common vulnerabilities

---

11. Success Metrics & KPIs

User Engagement Metrics (Primary)

Metric: Daily Active Users (DAU)

- **Target:** 30+ DAU after 4 weeks of beta launch
- **Measurement:** Count unique logins per day (track `lastLogin` timestamp)
- **Success Indicator:** If 30% of registered users (100 total) check dashboard daily

Metric: Average Session Duration

- **Target:** 3+ minutes per session
- **Measurement:** Time between login and last API call
- **Success Indicator:** Users spending time exploring sentiment analysis, not just quick price checks

Metric: Holdings per User

- **Target:** 8+ holdings average
- **Measurement:** `COUNT(holdings) / COUNT(users)`
- **Success Indicator:** Users treating it as primary portfolio tracker, not just testing

Metric: Feature Adoption Rate

- **Target:** 60% of users use sentiment analysis, 40% configure alerts
- **Measurement:** Track events: `sentiment_viewed`, `alert_configured`
- **Success Indicator:** Core differentiating features are valued

---

Technical Performance Metrics

Metric: API Response Time (95th Percentile)

- **Target:** <1.5 seconds for dashboard load
- **Measurement:** Log request duration, calculate p95 weekly
- **Success Indicator:** Fast enough to feel "real-time"

Metric: Uptime

- **Target:** 99% uptime (excluding scheduled maintenance)
- **Measurement:** Uptime monitoring service (UptimeRobot free tier)
- **Success Indicator:** Reliable enough for daily use

Metric: Sentiment Analysis Accuracy

- **Target:** 70%+ agreement with human labels
- **Measurement:** Sample 100 articles monthly, manually label, compare to model
- **Success Indicator:** Sentiment insights are trustworthy

Metric: Cache Hit Rate

- **Target:** 75% for price data, 90% for sentiment data
- **Measurement:** `(cache_hits / total_requests) * 100`
- **Success Indicator:** Effectively managing free-tier API limits

---

Portfolio Project Effectiveness (For Developers)

Metric: Technical Interview Mentions

- **Target:** 80% of users (developers) mention PortfolioPulse in interviews
- **Measurement:** Post-interview survey (Google Form)
- **Success Indicator:** Project serves as strong talking point

Metric: Code Quality Score

- **Target:** A grade on Code Climate or similar tool
- **Measurement:** Automated code analysis
- **Success Indicator:** Demonstrates professional coding standards

Metric: GitHub Stars (If Open-Sourced)

- **Target:** 50+ stars within 3 months
- **Measurement:** GitHub API
- **Success Indicator:** Project resonates with developer community

Metric: Architecture Complexity

- **Target:** Demonstrates 5+ architectural patterns (microservices, caching, rate limiting, hybrid APIs, ML integration)
- **Measurement:** Manual checklist
- **Success Indicator:** Showcases breadth of technical skills

---

Business Metrics (Secondary)

Metric: User Retention (Week 4)

- **Target:** 40% of Week 1 users still active in Week 4
- **Measurement:** Cohort analysis in MongoDB
- **Success Indicator:** Product has lasting value

Metric: Net Promoter Score (NPS)

- **Target:** 30+ (acceptable for MVP)
- **Measurement:** In-app survey: "How likely to recommend (0-10)?"
- **Success Indicator:** Users find it useful enough to share

---

12. Risks & Mitigation Strategies

Technical Risks

Risk 1: API Reliability & Rate Limiting

- **Probability:** High (70%)
- **Impact:** Critical (app unusable if price data unavailable)
- **Mitigation:**
  - Implement aggressive caching (5-minute TTL for prices)
  - Build Alpha Vantage fallback for Yahoo Finance failures
  - Display stale data with timestamp when rate limit hit
  - Pre-cache top 100 stocks proactively during market hours
- **Contingency:** If Yahoo Finance permanently breaks, pivot to Alpha Vantage as primary (accept 25 ticker limit for MVP)

Risk 2: AI Model Accuracy Below Expectations

- **Probability:** Medium (40%)
- **Impact:** High (key differentiator fails to deliver value)
- **Mitigation:**
  - Use pre-trained FinBERT (already 65% accurate on financial text)
  - Set user expectations: "AI-powered insights (beta)" disclaimer
  - Collect user feedback: "Was this sentiment accurate?" thumbs up/down
  - Manual review of 100 sentiment predictions weekly
- **Contingency:** If accuracy <50%, disable sentiment feature and pivot to manual news aggregation without scores

Risk 3: Database Performance Degradation

- **Probability:** Low (20%)
- **Impact:** Medium (slow dashboard loads hurt UX)
- **Mitigation:**
  - Comprehensive indexing strategy from day 1
  - Use MongoDB explain() to analyze slow queries during development
  - Implement pagination for large datasets (>100 holdings)
  - Set up database query monitoring (MongoDB Atlas built-in)
- **Contingency:** Upgrade to M10 tier ($57/month) if M0 free tier insufficient

Risk 4: Python Service Downtime

- **Probability:** Medium (30%)
- **Impact:** Medium (sentiment unavailable, but portfolio tracking still works)
- **Mitigation:**
  - Implement circuit breaker: After 10 failures, stop calling Python service for 5 minutes
  - Return neutral sentiment (0.0) as fallback
  - Display "Sentiment analysis temporarily unavailable" banner
  - Monitor Python service health with `/health` endpoint pings every 60 seconds
- **Contingency:** If Python service unreliable, pre-compute sentiment overnight via batch job instead of real-time

---

Development Risks

Risk 5: Scope Creep

- **Probability:** High (60%)

- **Impact:** High (missed 8-week deadline)
- **Mitigation:**
  - Strict P0/P1/P2 prioritization: Only P0 features in MVP
  - Weekly sprint reviews: "Are we on track for Sprint goals?"
  - Document all feature requests in "Post-MVP Backlog"
  - Set firm launch date: Week 8, no exceptions
- **Contingency:** If behind schedule, cut P1 features (email alerts, risk scoring) and launch with P0 only

Risk 6: Underestimated Complexity

- **Probability:** Medium (50%)
- **Impact:** High (delayed launch)
- **Mitigation:**
  - Add 20% buffer to all task estimates
  - Daily standups to identify blockers early
  - Pair programming for complex features (GraphQL schema, sentiment integration)
  - Week 9 buffer period for catching up
- **Contingency:** If week 9 insufficient, launch with known bugs marked as "Known Issues" and fix post-launch

Risk 7: External Dependency Changes

- **Probability:** Low (15%)
- **Impact:** Critical (Yahoo Finance schema change breaks app)
- **Mitigation:**
  - Abstract all API calls behind adapter layer (easy to swap)
  - Pin npm package versions (no ^ or ~ in package.json for external APIs)
  - Set up monitoring for API response schema changes
  - Keep Alpha Vantage integration ready as hot-swap replacement
- **Contingency:** If Yahoo Finance breaks, immediately switch to Alpha Vantage and communicate limitation (25 tickers max) to users

---

User Adoption Risks

Risk 8: Poor User Retention

- **Probability:** Medium (40%)
- **Impact:** Medium (hurts portfolio showcase narrative)
- **Mitigation:**
  - Launch with clear value proposition: "See why your portfolio moved today"
  - Implement email alerts to drive re-engagement
  - Add empty state prompts: "Add your first holding to get started"
  - Collect feedback via in-app survey after 1 week
- **Contingency:** If retention <20%, pivot messaging to focus on AI insights (may need better onboarding tutorial)

Risk 9: Insufficient Beta Testers

- **Probability:** Low (20%)
- **Impact:** Medium (less feedback for improvements)
- **Mitigation:**
  - Recruit from personal network, Reddit (r/reactjs, r/algotrading)
  - Offer early access in exchange for feedback
  - Create landing page with email signup 2 weeks before launch
- **Contingency:** If <20 beta users, launch publicly on Product Hunt / Hacker News to get traffic

---

Security Risks

Risk 10: Data Breach or Unauthorized Access

- **Probability:** Low (10%)
- **Impact:** Critical (damages reputation, legal liability)
- **Mitigation:**
  - Follow OWASP Top 10 guidelines from day 1
  - Use Helmet.js for security headers
  - Never log sensitive data (passwords, full JWT tokens)
  - Implement rate limiting on auth endpoints
  - Run automated security scan (Snyk free tier)
- **Contingency:** If breach occurs, immediately rotate all secrets, force password resets, notify users within 72 hours (GDPR requirement)

Conclusion

PortfolioPulse represents a unique intersection of practical utility and technical sophistication. By leveraging free-tier APIs, open-source AI models, and modern full-stack architecture, this project delivers enterprise-grade portfolio tracking at zero cost while serving as a compelling portfolio piece for developers.

Key Success Factors:

1. **Ruthless Prioritization:** Focus on P0 features (tracking, sentiment, auth) before polish
2. **API Rate Limit Management:** Caching and queuing are non-negotiable for free tiers
3. **Hybrid API Strategy:** REST for simplicity, GraphQL for complexity—pick the right tool
4. **AI as Enhancement:** Sentiment analysis differentiates but doesn't block core functionality
5. **Timeline Discipline:** 8 weeks is tight; use buffer week wisely

Next Steps:

- Engineering review of PRD (1 day)
- Environment setup: MongoDB Atlas, GitHub repo, local dev (1 day)
- Sprint 1 kickoff (Week 1, Day 1)

Document Approval:

- ☐ Product Manager Sign-Off
- ☐ Engineering Lead Review
- ☐ UX Designer Consultation (for dashboard mockups)
- ☐ Stakeholder Alignment (if applicable)

Appendix A: Glossary

- **FinBERT:** Financial BERT model pre-trained on financial text
- **HHI:** Herfindahl-Hirschman Index (concentration metric)
- **JWT:** JSON Web Token (stateless authentication)
- **TTL:** Time To Live (cache expiry duration)
- **P0/P1/P2:** Priority levels (0=critical, 1=important, 2=nice-to-have)

Appendix B: Additional Resources

- FinBERT Model: https://huggingface.co/ProsusAI/finbert

- Yahoo Finance API Docs: https://github.com/gadicc/node-yahoo-finance2
- GraphQL Best Practices: https://graphql.org/learn/best-practices/
- MongoDB Schema Design Patterns: https://www.mongodb.com/blog/post/building-with-patterns-a-summary

---

End of Product Requirements Document