BACKEND TEAM - DETAILED JIRA TASKS WITH PROFESSOR SPECIFICATIONS

Epic: Value Investing Backend Infrastructure

📋 SPRINT 1: Database Foundation & Core Services

BACK-001: Database Schema Implementation

Type: Backend Task
Priority: Highest
Story Points: 8
Assignee: Senior Backend Developer

Description: Implement the fundamental analysis database schema with proper indexes and constraints, including peer comparison fields.

Acceptance Criteria:

- [ ] Execute all SQL scripts for new tables
- [ ] Add peer comparison columns to existing `stocks` table
- [ ] Create all indexes for query performance
- [ ] Create helper views for efficient data access
- [ ] Test all foreign key constraints
- [ ] Document schema changes with migration scripts
- [ ] Verify backward compatibility with existing technical analysis

Technical Details:

```sql
-- Enhanced stocks table with peer data:
ALTER TABLE stocks ADD COLUMN
    peer_1_ticker VARCHAR(10),
    peer_2_ticker VARCHAR(10),
    peer_3_ticker VARCHAR(10),
    sector_etf_ticker VARCHAR(10),
    peers_last_updated TIMESTAMP,
    industry_classification VARCHAR(100),
    gics_sector VARCHAR(50),
    gics_industry VARCHAR(100),
    market_cap_category VARCHAR(20), -- 'large', 'mid', 'small', 'micro'
    fundamentals_last_update TIMESTAMP,
    next_earnings_date DATE,
    data_priority INTEGER DEFAULT 1;

-- Foreign key constraints:
ALTER TABLE stocks ADD CONSTRAINT fk_peer_1 FOREIGN KEY (peer_1_ticker) REFERENCES stocks(ticke
ALTER TABLE stocks ADD CONSTRAINT fk_peer_2 FOREIGN KEY (peer_2_ticker) REFERENCES stocks(ticke
ALTER TABLE stocks ADD CONSTRAINT fk_peer_3 FOREIGN KEY (peer_3_ticker) REFERENCES stocks(ticke

-- Tables to create:
- company_fundamentals (core financial data)
- financial_ratios (calculated metrics with all professor-specified ratios)
- industry_benchmarks (market-cap weighted averages)
- earnings_calendar (update priority system)
- api_usage_tracking (rate limit monitoring)
- investor_scores (cached calculations with 3 profiles)
```

Definition of Done:

- All tables created without errors
- Peer comparison constraints working
- Sample data can be inserted/queried
- Performance tests show <100ms for typical queries

BACK-002: API Rate Limiting & Usage Tracking Service

Type: Backend Task
Priority: High

Story Points: 13
Assignee: Backend Developer

Description: Build a centralized API usage tracking service to manage rate limits across multiple providers.

Acceptance Criteria:

- [ ] Track API calls per provider per day
- [ ] Implement rate limiting before hitting API limits
- [ ] Auto-reset counters at provider-specific times
- [ ] Queue system for delayed API calls
- [ ] Alert system when approaching limits
- [ ] Graceful degradation when limits exceeded

Rate Limits to Manage:

```python
API_LIMITS = {
    'yahoo': {'calls_per_hour': 2000, 'calls_per_day': 20000},
    'finnhub': {'calls_per_minute': 60, 'calls_per_day': 86400},
    'alphavantage': {'calls_per_minute': 5, 'calls_per_day': 500}
}
```

Code Structure:

```python
class APIRateLimiter:
    def check_limit(self, provider: str, endpoint: str) -> bool
    def record_call(self, provider: str, endpoint: str) -> None
    def get_next_available_time(self, provider: str) -> datetime
    def queue_request(self, provider: str, request_data: dict) -> None
```

BACK-003: Yahoo Finance API Integration Service

Type: Backend Task
Priority: High
Story Points: 13
Assignee: Backend Developer

Description: Implement Yahoo Finance API integration with comprehensive error handling and data parsing.

Acceptance Criteria:

- [ ] Fetch financial statements (income, balance sheet, cash flow)
- [ ] Parse key metrics and ratios
- [ ] Handle missing data gracefully
- [ ] Implement exponential backoff retry logic
- [ ] Standardize data format for database storage
- [ ] Log all API responses for debugging

Data Points to Fetch:

```python
YAHOO_ENDPOINTS = {
    'financials': '/v1/finance/financials',
    'key_statistics': '/v11/finance/quoteSummary',
    'balance_sheet': '/v1/finance/balance_sheet',
    'cash_flow': '/v1/finance/cash_flow'
}
```

Error Handling:

- HTTP timeouts after 30 seconds
- Retry 3 times with 2^attempt delay
- Fallback to cached data if available
- Log errors to `data_update_log` table

BACK-004: Finnhub Backup API Service

Type: Backend Task
Priority: Medium
Story Points: 8
Assignee: Backend Developer

Description: Implement Finnhub API as secondary data source when Yahoo Finance fails.

Acceptance Criteria:

- [ ] Map Finnhub data fields to our schema

- [ ] Implement as fallback service
- [ ] Handle free tier limitations (60 calls/min)
- [ ] Parse financial data from different JSON structure
- [ ] Quality scoring for data completeness

API Mapping:

```python
FINNHUB_MAPPING = {
    'revenue': 'reportedCurrency.totalRevenue',
    'net_income': 'reportedCurrency.netIncome',
    'total_debt': 'reportedCurrency.totalDebt',
    # ... additional mappings
}
```

BACK-005: Alpha Vantage Tertiary Service

Type: Backend Task
Priority: Low
Story Points: 5
Assignee: Junior Backend Developer

Description: Implement Alpha Vantage as final fallback for fundamental data.

Acceptance Criteria:

- [ ] Handle extremely low rate limits (5 calls/min)
- [ ] Focus on annual data only due to limitations
- [ ] Aggressive caching (cache for 24 hours minimum)
- [ ] Use only for companies with no other data sources

📋 SPRINT 2: Smart Data Pipeline & Calculations

BACK-006: Priority-Based Update Scheduler

Type: Backend Task
Priority: Highest
Story Points: 21
Assignee: Senior Backend Developer

Description: Build intelligent scheduling system that prioritizes companies based on earnings dates and data freshness.

Acceptance Criteria:

- [ ] Daily cron job identifies companies needing updates
- [ ] Priority algorithm: earnings (5), no data (4), 30+ days (3), 90+ days (2), rest (1)
- [ ] Distribute API calls across multiple days to respect rate limits
- [ ] Track success/failure rates per company
- [ ] Retry failed updates with exponential backoff

Priority Algorithm:

```python
def calculate_priority(ticker_data):
    if ticker_data.next_earnings_date <= today + 7_days:
        return 5  # Highest - earnings soon
    elif ticker_data.fundamentals_last_update is None:
        return 4  # High - no data
    elif ticker_data.fundamentals_last_update < today - 30_days:
        return 3  # Medium - stale data
    elif ticker_data.fundamentals_last_update < today - 90_days:
        return 2  # Low - old data
    else:
        return 1  # Lowest - recent data
```

Daily Limits:

- Monday: 400 companies (Yahoo heavy day)
- Tuesday: 300 companies (Finnhub focused)
- Wednesday: 400 companies (Yahoo continued)
- Thursday: 300 companies (catch-up day)
- Friday: 100 companies (Alpha Vantage only)

BACK-007A: Core Financial Ratios Calculator

Type: Backend Task
Priority: High
Story Points: 13
Assignee: Senior Backend Developer

Description: Implement exact financial ratio calculations per professor specifications with precise edge case handling.

Acceptance Criteria:

- [ ] Implement all valuation ratios with exact formulas
- [ ] Handle negative values and edge cases precisely
- [ ] Add data quality flags for each calculation
- [ ] Unit tests covering all edge cases
- [ ] Performance optimization for bulk calculations

EXACT FORMULAS TO IMPLEMENT:

Valuation Ratios:

python

```python
def calculate_pe_ratio(self, current_price: float, diluted_eps_ttm: float) -> Optional[float]:
    """
    P/E = Current Market Price / Diluted EPS (TTM)
    - Use diluted EPS for conservative calculation
    - If EPS ≤ 0, return None and flag as "N/A - Negative Earnings"
    - Cap display at 999 to handle extreme cases
    """
    if diluted_eps_ttm <= 0:
        return None  # Flag: "N/A - Negative Earnings"

    pe_ratio = current_price / diluted_eps_ttm
    return min(pe_ratio, 999)  # Cap extreme values

def calculate_pb_ratio(self, market_cap: float, shareholders_equity: float) -> Optional[float]:
    """
    P/B = Market Capitalization / Total Shareholders' Equity
    - Use most recent quarterly shareholders' equity
    - If book value ≤ 0, return None and flag as "N/A - Negative Book Value"
    """
    if shareholders_equity <= 0:
        return None  # Flag: "N/A - Negative Book Value"

    return market_cap / shareholders_equity

def calculate_ev_ebitda(self, market_cap: float, total_debt: float,
                        cash: float, ebitda_ttm: float) -> Optional[float]:
    """
    Enterprise Value = Market Cap + Total Debt - Cash and Cash Equivalents
    EV/EBITDA = Enterprise Value / EBITDA (TTM)
    - If EBITDA ≤ 0, return None and flag as "N/A - Negative EBITDA"
    """
    if ebitda_ttm <= 0:
        return None  # Flag: "N/A - Negative EBITDA"

    enterprise_value = market_cap + total_debt - cash
    return enterprise_value / ebitda_ttm

def calculate_ps_ratio(self, market_cap: float, revenue_ttm: float) -> Optional[float]:
    """
    P/S = Market Capitalization / Revenue (TTM)
    - Always use TTM revenue for recency
    - Most reliable for loss-making companies
    - Cap at 50 for display purposes
```

```
    """
    if revenue_ttm <= 0:
        return None

    ps_ratio = market_cap / revenue_ttm
    return min(ps_ratio, 50)  # Cap at 50

def calculate_peg_ratio(self, pe_ratio: float, earnings_growth_3yr: float) -> Optional[float]:
    """
    PEG = P/E Ratio / Earnings Growth Rate (3-year CAGR)
    - Use 3-year historical earnings CAGR as baseline
    - If growth rate ≤ 0, return None
    - If PEG > 5, flag as "High Growth Premium"
    """
    if pe_ratio is None or earnings_growth_3yr <= 0:
        return None

    peg = pe_ratio / (earnings_growth_3yr * 100)  # Convert percentage to decimal
    if peg > 5:
        # Flag: "High Growth Premium"
        pass
    return peg
```

Edge Case Documentation:

- All functions return None for invalid inputs
- Flags are stored in separate data quality field
- Cap extreme values to prevent display issues
- Log all edge cases for monitoring

BACK-007B: Graham Number & Altman Z-Score Calculator

Type: Backend Task
Priority: High
Story Points: 8
Assignee: Backend Developer

Description: Implement Graham Number and Altman Z-Score calculations with exact professor specifications.

Acceptance Criteria:

- [ ] Classic Graham Number formula implementation
- [ ] Original Altman Z-Score for public companies
- [ ] Special sector adjustments for Z-Score
- [ ] Proper handling of negative inputs
- [ ] Risk zone classification

EXACT FORMULAS:

Graham Number:

```python
def calculate_graham_number(self, diluted_eps_ttm: float, book_value_per_share: float) -> Optic
    """
    Graham Number = √(15 × Diluted EPS (TTM) × Book Value per Share)

    Where:
    - 15 = Maximum P/E ratio for defensive investors
    - EPS must be positive (if negative, return None)
    - BVPS must be positive (if negative, return None)
    """
    if diluted_eps_ttm <= 0 or book_value_per_share <= 0:
        return None  # Flag: "N/A - Requires Positive Earnings & Book Value"

    graham_number = math.sqrt(15 * diluted_eps_ttm * book_value_per_share)
    return graham_number

def get_graham_number_note(self, industry: str) -> str:
    """
    For intangible-heavy businesses (tech), provide adjusted note
    """
    intangible_heavy_industries = ['Software', 'Internet', 'Biotechnology']
    if industry in intangible_heavy_industries:
        return "Note: Graham Number may undervalue companies with significant intangible assets
    return ""
```

Altman Z-Score:

python

```python
def calculate_altman_z_score(self, working_capital: float, total_assets: float,
                             retained_earnings: float, ebit: float,
                             market_value_equity: float, total_liabilities: float,
                             sales: float) -> float:
    """
    Z-Score = 1.2(A) + 1.4(B) + 3.3(C) + 0.6(D) + 1.0(E)

    Where:
    A = Working Capital / Total Assets
    B = Retained Earnings / Total Assets
    C = EBIT / Total Assets
    D = Market Value of Equity / Total Liabilities
    E = Sales / Total Assets
    """
    if total_assets <= 0 or total_liabilities <= 0:
        return 0.0  # Invalid data

    A = working_capital / total_assets
    B = retained_earnings / total_assets
    C = ebit / total_assets
    D = market_value_equity / total_liabilities
    E = sales / total_assets

    z_score = 1.2 * A + 1.4 * B + 3.3 * C + 0.6 * D + 1.0 * E
    return z_score

def classify_z_score_risk(self, z_score: float) -> dict:
    """
    Interpretation:
    - Z > 2.99: Safe Zone (Low bankruptcy risk)
    - 1.81 < Z < 2.99: Gray Zone (Moderate risk)
    - Z < 1.81: Distress Zone (High bankruptcy risk)
    """
    if z_score > 2.99:
        return {"zone": "Safe", "risk": "Low", "color": "green"}
    elif z_score >= 1.81:
        return {"zone": "Gray", "risk": "Moderate", "color": "yellow"}
    else:
        return {"zone": "Distress", "risk": "High", "color": "red"}
```

BACK-007C: Profitability & Quality Ratios Calculator

Type: Backend Task
Priority: High
Story Points: 8
Assignee: Backend Developer

Description: Implement ROE, ROIC, ROA and quality metrics with exact professor specifications.

Acceptance Criteria:

- [ ] ROE with average equity calculation
- [ ] ROIC with NOPAT methodology
- [ ] ROA with average assets
- [ ] Quality thresholds implementation
- [ ] DuPont analysis components

EXACT FORMULAS:

python

```python
def calculate_roe(self, net_income_ttm: float, equity_quarters: List[float]) -> Optional[float]
    """
    ROE = Net Income (TTM) / Average Shareholders' Equity
    - Use average of last 4 quarters' equity for stability
    - Express as percentage
    - Flag ROE > 30% as "Exceptional - Verify Sustainability"
    """
    if len(equity_quarters) < 2:
        return None

    avg_equity = sum(equity_quarters) / len(equity_quarters)
    if avg_equity <= 0:
        return None

    roe = (net_income_ttm / avg_equity) * 100

    if roe > 30:
        # Flag: "Exceptional - Verify Sustainability"
        pass

    return roe

def calculate_roic(self, operating_income: float, tax_rate: float,
                   total_assets: float, cash: float, current_liabilities: float,
                   interest_bearing_debt: float) -> Optional[float]:
    """
    ROIC = NOPAT / Invested Capital

    Where:
    NOPAT = Operating Income × (1 - Effective Tax Rate)
    Invested Capital = Total Assets - Cash - Non-Interest Bearing Current Liabilities
    """
    nopat = operating_income * (1 - tax_rate)

    # Non-interest bearing current liabilities (approximate as current liabilities - interest b
    non_interest_bearing_cl = current_liabilities - min(current_liabilities, interest_bearing_d
    invested_capital = total_assets - cash - non_interest_bearing_cl

    if invested_capital <= 0:
        return None

    roic = (nopat / invested_capital) * 100
    return roic
```

```python
def classify_roic_quality(self, roic: float) -> str:
    """
    Quality Threshold:
    - ROIC > 15%: Excellent
    - ROIC 10-15%: Good
    - ROIC 5-10%: Average
    - ROIC < 5%: Poor
    """
    if roic > 15:
        return "Excellent"
    elif roic >= 10:
        return "Good"
    elif roic >= 5:
        return "Average"
    else:
        return "Poor"

def calculate_roa(self, net_income_ttm: float, assets_quarters: List[float]) -> Optional[float]
    """
    ROA = Net Income (TTM) / Average Total Assets
    - Use average total assets over 4 quarters
    - Key indicator of management efficiency
    """
    if len(assets_quarters) < 2:
        return None

    avg_assets = sum(assets_quarters) / len(assets_quarters)
    if avg_assets <= 0:
        return None

    roa = (net_income_ttm / avg_assets) * 100
    return roa
```

BACK-008: Investor Scoring System

Type: Backend Task

Priority: High

Story Points: 13

Assignee: Backend Developer

Description: Implement the three investor profile scoring systems with weighted components.

Acceptance Criteria:

- [ ] Calculate scores for Conservative, GARP, Deep Value profiles
- [ ] Component-level scoring for breakdown display
- [ ] Risk warning system integration
- [ ] Score explanation generation
- [ ] Cache results in `investor_scores` table

Scoring Weights:

```python
INVESTOR_WEIGHTS = {
    'conservative': {
        'financial_health': 0.30,
        'valuation': 0.25,
        'quality': 0.20,
        'profitability': 0.15,
        'growth': 0.05,
        'management': 0.05
    },
    'garp': {
        'valuation': 0.25,
        'growth': 0.25,
        'quality': 0.20,
        'profitability': 0.15,
        'financial_health': 0.10,
        'management': 0.05
    },
    'deep_value': {
        'valuation': 0.40,
        'financial_health': 0.25,
        'quality': 0.15,
        'profitability': 0.10,
        'management': 0.05,
        'growth': 0.05
    }
}
```

BACK-009A: Peer Selection Algorithm Implementation

Type: Backend Task

Priority: High

Story Points: 13
Assignee: Backend Developer

Description: Implement algorithmic peer selection system using professor's exact similarity scoring methodology.

Acceptance Criteria:

- [ ] Implement SQL-based peer selection algorithm
- [ ] Calculate similarity scores based on industry, market cap, and revenue
- [ ] Populate peer fields in stocks table
- [ ] Link sector ETFs from market_etf table
- [ ] Weekly peer refresh mechanism

EXACT PEER SELECTION ALGORITHM:

sql

```sql
-- Professor's Algorithmic Selection (Implemented in SQL)
WITH peer_candidates AS (
  SELECT
    target.ticker as company_ticker,
    candidate.ticker as peer_ticker,
    -- Similarity scoring
    CASE
      WHEN target.industry = candidate.industry THEN 30
      WHEN target.sector = candidate.sector THEN 15
      ELSE 0
    END +
    CASE
      WHEN ABS(LOG(target.market_cap) - LOG(candidate.market_cap)) < 0.5 THEN 25
      WHEN ABS(LOG(target.market_cap) - LOG(candidate.market_cap)) < 1.0 THEN 15
      WHEN ABS(LOG(target.market_cap) - LOG(candidate.market_cap)) < 1.5 THEN 5
      ELSE 0
    END +
    CASE
      WHEN ABS(LOG(COALESCE(target.revenue, target.market_cap * 0.1)) -
               LOG(COALESCE(candidate.revenue, candidate.market_cap * 0.1))) < 0.5 THEN 20
      WHEN ABS(LOG(COALESCE(target.revenue, target.market_cap * 0.1)) -
               LOG(COALESCE(candidate.revenue, candidate.market_cap * 0.1))) < 1.0 THEN 10
      ELSE 0
    END as similarity_score
  FROM stocks target
  CROSS JOIN stocks candidate
  WHERE target.ticker != candidate.ticker
    AND candidate.market_cap IS NOT NULL
    AND candidate.market_cap > 100000000  -- Min $100M market cap
),
ranked_peers AS (
  SELECT
    company_ticker,
    peer_ticker,
    similarity_score,
    ROW_NUMBER() OVER (PARTITION BY company_ticker ORDER BY similarity_score DESC) as peer_rank
  FROM peer_candidates
  WHERE similarity_score >= 40  -- Minimum similarity threshold
)
UPDATE stocks s SET
  peer_1_ticker = (SELECT peer_ticker FROM ranked_peers WHERE company_ticker = s.ticker AND pee
  peer_2_ticker = (SELECT peer_ticker FROM ranked_peers WHERE company_ticker = s.ticker AND pee
  peer_3_ticker = (SELECT peer_ticker FROM ranked_peers WHERE company_ticker = s.ticker AND pee
```

```
  peers_last_updated = CURRENT_TIMESTAMP
WHERE EXISTS (SELECT 1 FROM ranked_peers WHERE company_ticker = s.ticker);

-- Link sector ETFs
UPDATE stocks s SET sector_etf_ticker = (
  SELECT me.etf_ticker
  FROM market_etf me
  WHERE UPPER(me.category) = UPPER(s.sector)
    OR UPPER(me.indicator) = UPPER(s.sector)
  LIMIT 1
);
```

Python Implementation:

python

```python
class PeerSelectionService:
    def calculate_peer_similarity(self, target_company: dict, candidate: dict) -> float:
        """
        Calculate similarity score between two companies
        Max score: 75 points (30 + 25 + 20)
        """
        score = 0

        # Industry/Sector match (30 points max)
        if target_company['industry'] == candidate['industry']:
            score += 30
        elif target_company['sector'] == candidate['sector']:
            score += 15

        # Market cap similarity (25 points max)
        if target_company['market_cap'] and candidate['market_cap']:
            market_cap_ratio = abs(
                math.log(target_company['market_cap']) -
                math.log(candidate['market_cap'])
            )
            if market_cap_ratio < 0.5:
                score += 25
            elif market_cap_ratio < 1.0:
                score += 15
            elif market_cap_ratio < 1.5:
                score += 5

        # Revenue similarity (20 points max)
        target_revenue = target_company.get('revenue') or target_company['market_cap'] * 0.1
        candidate_revenue = candidate.get('revenue') or candidate['market_cap'] * 0.1

        if target_revenue and candidate_revenue:
            revenue_ratio = abs(math.log(target_revenue) - math.log(candidate_revenue))
            if revenue_ratio < 0.5:
                score += 20
            elif revenue_ratio < 1.0:
                score += 10

        return score

    def select_top_peers(self, ticker: str) -> List[str]:
        """
        Select top 3 peers for a given company
```

```
    """
    target_company = self.get_company_data(ticker)
    candidates = self.get_all_companies(exclude=ticker)

    peer_scores = []
    for candidate in candidates:
        if candidate['market_cap'] >= 100_000_000:  # Min $100M
            score = self.calculate_peer_similarity(target_company, candidate)
            if score >= 40:  # Minimum threshold
                peer_scores.append((candidate['ticker'], score))

    # Sort by score and return top 3
    peer_scores.sort(key=lambda x: x[1], reverse=True)
    return [ticker for ticker, score in peer_scores[:3]]
```

BACK-009B: Industry Benchmarking Service

Type: Backend Task
Priority: Medium
Story Points: 13
Assignee: Backend Developer

Description: Calculate industry averages and percentiles using professor's market-cap weighted methodology.

Acceptance Criteria:

- [ ] Weekly industry benchmark calculations
- [ ] Market-cap weighted averages (not simple averages)
- [ ] Calculate 25th, 50th, 75th, 90th percentiles
- [ ] Handle insufficient peer data with sector fallback
- [ ] Store results in industry_benchmarks table

EXACT BENCHMARKING METHODOLOGY:

Market-Cap Weighted Approach:

sql

```sql
-- Industry benchmark calculation per professor specifications
WITH industry_metrics AS (
  SELECT
    s.industry,
    s.sector,
    COUNT(*) as company_count,

    -- Market-cap weighted averages (not simple averages)
    SUM(fr.pe_ratio * s.market_cap) / NULLIF(SUM(s.market_cap), 0) as weighted_avg_pe,
    SUM(fr.pb_ratio * s.market_cap) / NULLIF(SUM(s.market_cap), 0) as weighted_avg_pb,
    SUM(fr.roe * s.market_cap) / NULLIF(SUM(s.market_cap), 0) as weighted_avg_roe,
    SUM(fr.debt_to_equity * s.market_cap) / NULLIF(SUM(s.market_cap), 0) as weighted_avg_debt_e

    -- Percentiles (for distribution understanding)
    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY fr.pe_ratio) as pe_25th,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY fr.pe_ratio) as pe_median,
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY fr.pe_ratio) as pe_75th,
    PERCENTILE_CONT(0.90) WITHIN GROUP (ORDER BY fr.pe_ratio) as pe_90th,

    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY fr.roe) as roe_25th,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY fr.roe) as roe_median,
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY fr.roe) as roe_75th,
    PERCENTILE_CONT(0.90) WITHIN GROUP (ORDER BY fr.roe) as roe_90th,

    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY fr.debt_to_equity) as debt_eq_25th,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY fr.debt_to_equity) as debt_eq_median,
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY fr.debt_to_equity) as debt_eq_75th,
    PERCENTILE_CONT(0.90) WITHIN GROUP (ORDER BY fr.debt_to_equity) as debt_eq_90th

  FROM financial_ratios fr
  JOIN stocks s ON fr.ticker = s.ticker
  WHERE fr.calculation_date = (
    SELECT MAX(calculation_date) FROM financial_ratios fr2 WHERE fr2.ticker = fr.ticker
  )
    AND fr.pe_ratio IS NOT NULL
    AND fr.pe_ratio BETWEEN 0 AND 100  -- Exclude outliers
    AND s.market_cap > 100000000  -- Min $100M market cap
  GROUP BY s.industry, s.sector
),
-- Handle insufficient industry data with sector fallback
sector_fallback AS (
  SELECT
    s.sector,
```

```
    COUNT(*) as sector_company_count,
    SUM(fr.pe_ratio * s.market_cap) / NULLIF(SUM(s.market_cap), 0) as sector_weighted_avg_pe,
    SUM(fr.pb_ratio * s.market_cap) / NULLIF(SUM(s.market_cap), 0) as sector_weighted_avg_pb,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY fr.pe_ratio) as sector_pe_median
  FROM financial_ratios fr
  JOIN stocks s ON fr.ticker = s.ticker
  WHERE fr.calculation_date = (
    SELECT MAX(calculation_date) FROM financial_ratios fr2 WHERE fr2.ticker = fr.ticker
  )
  GROUP BY s.sector
  HAVING COUNT(*) >= 10  -- Minimum 10 companies for sector
)
INSERT INTO industry_benchmarks (
  industry_code, sector_code, calculation_date, company_count,
  avg_pe_ratio, median_pe_ratio, avg_pb_ratio, avg_roe, avg_debt_to_equity,
  pe_ratio_25p, pe_ratio_75p, pe_ratio_90p,
  roe_25p, roe_75p, roe_90p,
  debt_eq_25p, debt_eq_75p, debt_eq_90p
)
SELECT
  COALESCE(im.industry, 'SECTOR_' || im.sector) as industry_code,
  im.sector as sector_code,
  CURRENT_DATE as calculation_date,
  CASE
    WHEN im.company_count >= 5 THEN im.company_count
    ELSE sf.sector_company_count
  END as company_count,

  -- Use industry data if sufficient, otherwise sector data
  CASE
    WHEN im.company_count >= 5 THEN im.weighted_avg_pe
    ELSE sf.sector_weighted_avg_pe
  END as avg_pe_ratio,

  CASE
    WHEN im.company_count >= 5 THEN im.pe_median
    ELSE sf.sector_pe_median
  END as median_pe_ratio,

  -- Continue for other ratios...
  im.weighted_avg_pb, im.weighted_avg_roe, im.weighted_avg_debt_eq,
  im.pe_25th, im.pe_75th, im.pe_90th,
  im.roe_25th, im.roe_75th, im.roe_90th,
  im.debt_eq_25th, im.debt_eq_75th, im.debt_eq_90th
```

```
FROM industry_metrics im
LEFT JOIN sector_fallback sf ON im.sector = sf.sector
WHERE im.company_count >= 5 OR sf.sector_company_count >= 10;
```

Python Service Implementation:

python

```python
class IndustryBenchmarkService:
    def calculate_industry_benchmarks(self) -> None:
        """
        Weekly calculation of industry benchmarks using market-cap weighting
        """
        industries = self.get_industries_with_sufficient_data()

        for industry in industries:
            companies = self.get_companies_by_industry(industry['code'])

            if len(companies) >= 5:
                benchmarks = self.calculate_weighted_benchmarks(companies)
                self.save_industry_benchmarks(industry['code'], benchmarks)
            else:
                # Fallback to sector level
                sector_companies = self.get_companies_by_sector(industry['sector'])
                if len(sector_companies) >= 10:
                    benchmarks = self.calculate_weighted_benchmarks(sector_companies)
                    self.save_industry_benchmarks(
                        f"SECTOR_{industry['sector']}",
                        benchmarks
                    )

    def calculate_weighted_benchmarks(self, companies: List[dict]) -> dict:
        """
        Calculate market-cap weighted averages and percentiles
        """
        total_market_cap = sum(c['market_cap'] for c in companies if c['market_cap'])

        # Weighted averages
        weighted_pe = sum(
            c['pe_ratio'] * c['market_cap']
            for c in companies
            if c['pe_ratio'] and c['market_cap']
        ) / total_market_cap if total_market_cap > 0 else None

        # Percentiles (unweighted for distribution understanding)
        pe_values = [c['pe_ratio'] for c in companies if c['pe_ratio']]
        percentiles = {}
        if pe_values:
            percentiles['pe_25th'] = numpy.percentile(pe_values, 25)
            percentiles['pe_median'] = numpy.percentile(pe_values, 50)
            percentiles['pe_75th'] = numpy.percentile(pe_values, 75)
```

```
            percentiles['pe_90th'] = numpy.percentile(pe_values, 90)

        return {
            'weighted_avg_pe': weighted_pe,
            'company_count': len(companies),
            **percentiles
        }

    def get_benchmark_for_company(self, ticker: str) -> dict:
        """
        Get appropriate benchmark for company (industry -> sector -> broad market)
        """
        company = self.get_company(ticker)

        # Try industry first
        industry_benchmark = self.get_industry_benchmark(company['industry'])
        if industry_benchmark and industry_benchmark['company_count'] >= 5:
            return industry_benchmark

        # Fallback to sector
        sector_benchmark = self.get_sector_benchmark(company['sector'])
        if sector_benchmark and sector_benchmark['company_count'] >= 10:
            return sector_benchmark

        # Final fallback to broad market
        return self.get_broad_market_benchmark(company['market_cap_category'])
```

Comparison Methodology Implementation:

python

```python
def calculate_peer_comparison_score(self, company_ticker: str) -> dict:
    """
    Weight: 70% peer comparison, 30% sector ETF comparison
    """
    company_ratios = self.get_company_ratios(company_ticker)

    # Get 3 peers
    peers = self.get_company_peers(company_ticker)
    peer_ratios = [self.get_company_ratios(peer) for peer in peers if peer]

    # Calculate peer averages
    peer_avg_pe = self.safe_average([r['pe_ratio'] for r in peer_ratios])
    peer_avg_pb = self.safe_average([r['pb_ratio'] for r in peer_ratios])

    # Score vs peers (higher is better for company, lower for ratios)
    peer_pe_score = self.score_ratio_comparison(
        company_ratios['pe_ratio'], peer_avg_pe, lower_is_better=True
    )
    peer_pb_score = self.score_ratio_comparison(
        company_ratios['pb_ratio'], peer_avg_pb, lower_is_better=True
    )

    # Sector ETF comparison (30% weight)
    sector_etf = self.get_sector_etf_ratios(company_ticker)
    etf_pe_score = self.score_ratio_comparison(
        company_ratios['pe_ratio'], sector_etf['pe_ratio'], lower_is_better=True
    )

    # Combined score
    final_score = (
        (peer_pe_score + peer_pb_score) / 2 * 0.7 +  # 70% peer
        etf_pe_score * 0.3  # 30% sector ETF
    )

    return {
        'overall_score': final_score,
        'peer_comparison': {
            'pe_score': peer_pe_score,
            'pb_score': peer_pb_score,
            'peers_used': peers
        },
        'sector_etf_comparison': {
            'etf_ticker': sector_etf['ticker'],
```

```
            'etf_score': etf_pe_score
        }
    }

def score_ratio_comparison(self, company_ratio: float, benchmark: float,
                           lower_is_better: bool = True) -> float:
    """
    Score company ratio vs benchmark
    Returns 0-100 score
    """
    if not company_ratio or not benchmark:
        return 50  # Neutral score for missing data

    ratio = company_ratio / benchmark

    if lower_is_better:
        # For P/E, P/B ratios - lower is better
        if ratio < 0.5:
            return 100  # Significantly better (cheaper)
        elif ratio < 0.75:
            return 75
        elif ratio < 1.0:
            return 60
        elif ratio < 1.25:
            return 40
        elif ratio < 1.5:
            return 25
        else:
            return 0  # Significantly worse (expensive)
    else:
        # For ROE, margins - higher is better
        if ratio > 2.0:
            return 100
        elif ratio > 1.5:
            return 75
        elif ratio > 1.25:
            return 60
        elif ratio > 1.0:
            return 50
        elif ratio > 0.75:
            return 25
        else:
            return 0
```

📋 SPRINT 3: Data Pipeline Automation & Monitoring

BACK-010: Earnings Calendar Integration

Type: Backend Task
Priority: High
Story Points: 13
Assignee: Backend Developer

Description: Maintain updated earnings calendar to prioritize fundamental data updates.

Acceptance Criteria:

- [ ] Fetch earnings dates from Yahoo Finance calendar
- [ ] Prioritize companies with earnings in next 7 days
- [ ] Flag companies for immediate update post-earnings
- [ ] Handle earnings date changes and confirmations
- [ ] Set update priority levels automatically

Update Trigger Logic:

```python
def update_earnings_priorities():
    # Mark companies with earnings this week as high priority
    upcoming = get_earnings_next_7_days()
    for ticker in upcoming:
        set_priority(ticker, level=5)

    # Reset priority for companies 30 days post-earnings
    old_earnings = get_earnings_older_than_30_days()
    for ticker in old_earnings:
        set_priority(ticker, level=1)
```

BACK-011A: Risk Warning System Implementation

Type: Backend Task
Priority: High
Story Points: 8
Assignee: Backend Developer

Description: Implement risk warning system with professor's exact threshold specifications and multi-factor triggering logic.

Acceptance Criteria:

- [ ] Implement exact warning thresholds from professor
- [ ] Multi-factor risk aggregation logic
- [ ] Real-time warning calculation
- [ ] Warning level escalation rules
- [ ] Integration with investor scoring system

EXACT WARNING THRESHOLDS (Per Professor):

python

```python
class RiskWarningSystem:

    # Professor's exact risk thresholds
    HIGH_RISK_THRESHOLDS = {
        # Financial Distress (any one triggers high risk)
        'altman_z_score': 1.8,  # Below this = high risk
        'interest_coverage': 1.5,  # Below this = high risk
        'current_ratio': 0.8,  # Below this = high risk
        'debt_to_equity': 3.0,  # Above this = high risk

        # Valuation Extreme (both conditions must be met)
        'pe_vs_industry_multiple': 3.0,  # AND pe_absolute > 50
        'pe_absolute_threshold': 50,
        'ev_ebitda_vs_industry_multiple': 3.0,  # AND ev_ebitda_absolute > 30
        'ev_ebitda_absolute_threshold': 30,

        # Quality Deterioration (progressive conditions)
        'roe_declining_years': 3,  # 3 consecutive years
        'fcf_negative_years': 2,   # 2+ years negative FCF
        'margin_decline_threshold': 0.20  # >20% decline over 3 years
    }

    WARNING_THRESHOLDS = {
        # Moderate Risk
        'altman_z_score_range': (1.8, 2.6),
        'interest_coverage_range': (1.5, 3.0),
        'debt_to_equity_range': (1.5, 3.0),
        'pe_vs_industry_multiple': 2.0
    }

    CAUTION_THRESHOLDS = {
        # Mild Concerns
        'altman_z_score_range': (2.6, 3.0),
        'interest_coverage_range': (3.0, 5.0),
        'debt_to_equity_range': (1.0, 1.5)
    }

    def evaluate_risk_warnings(self, ticker: str) -> dict:
        """
        Evaluate all risk factors and return warning level
        """
        company_data = self.get_company_financial_data(ticker)
        historical_data = self.get_historical_data(ticker, years=5)
```

```
        industry_data = self.get_industry_benchmarks(ticker)

        high_risk_factors = []
        warning_factors = []
        caution_factors = []

        # 🔴 HIGH RISK CHECKS
        high_risk_factors.extend(self._check_financial_distress(company_data))
        high_risk_factors.extend(self._check_valuation_extremes(company_data, industry_data))
        high_risk_factors.extend(self._check_quality_deterioration(historical_data))

        # 🟠 WARNING CHECKS
        if not high_risk_factors:  # Only check if no high risk
            warning_factors.extend(self._check_moderate_risks(company_data, industry_data))

        # 🟡 CAUTION CHECKS
        if not high_risk_factors and not warning_factors:
            caution_factors.extend(self._check_mild_concerns(company_data, historical_data))

        # Determine final warning level
        if high_risk_factors:
            warning_level = "high_risk"
            warning_icon = "🔴"
            primary_factors = high_risk_factors
        elif warning_factors:
            warning_level = "warning"
            warning_icon = "🟠"
            primary_factors = warning_factors
        elif caution_factors:
            warning_level = "caution"
            warning_icon = "🟡"
            primary_factors = caution_factors
        else:
            warning_level = "none"
            warning_icon = ""
            primary_factors = []

        return {
            'warning_level': warning_level,
            'warning_icon': warning_icon,
            'risk_factors': primary_factors,
            'factor_count': len(primary_factors),
            'detailed_analysis': {
                'high_risk': high_risk_factors,
```

```
            'warning': warning_factors,
            'caution': caution_factors
        }
    }

def _check_financial_distress(self, data: dict) -> List[str]:
    """Check for financial distress indicators"""
    factors = []

    if data.get('altman_z_score', 999) < self.HIGH_RISK_THRESHOLDS['altman_z_score']:
        factors.append(f"Altman Z-Score {data['altman_z_score']:.2f} indicates high bankrup

    if data.get('interest_coverage', 999) < self.HIGH_RISK_THRESHOLDS['interest_coverage']:
        factors.append(f"Interest coverage {data['interest_coverage']:.1f}x is dangerously

    if data.get('current_ratio', 999) < self.HIGH_RISK_THRESHOLDS['current_ratio']:
        factors.append(f"Current ratio {data['current_ratio']:.2f} suggests liquidity probl

    if data.get('debt_to_equity', 0) > self.HIGH_RISK_THRESHOLDS['debt_to_equity']:
        factors.append(f"Debt-to-equity {data['debt_to_equity']:.1f} is extremely high")

    return factors

def _check_valuation_extremes(self, data: dict, industry: dict) -> List[str]:
    """Check for extreme valuation metrics"""
    factors = []

    # P/E extreme check (both conditions must be met)
    pe_ratio = data.get('pe_ratio')
    industry_pe = industry.get('median_pe_ratio')

    if (pe_ratio and industry_pe and
        pe_ratio > industry_pe * self.HIGH_RISK_THRESHOLDS['pe_vs_industry_multiple'] and
        pe_ratio > self.HIGH_RISK_THRESHOLDS['pe_absolute_threshold']):
        factors.append(f"P/E {pe_ratio:.1f} is {pe_ratio/industry_pe:.1f}x industry median"

    # EV/EBITDA extreme check
    ev_ebitda = data.get('ev_ebitda')
    industry_ev_ebitda = industry.get('median_ev_ebitda')

    if (ev_ebitda and industry_ev_ebitda and
        ev_ebitda > industry_ev_ebitda * self.HIGH_RISK_THRESHOLDS['ev_ebitda_vs_industry_m
        ev_ebitda > self.HIGH_RISK_THRESHOLDS['ev_ebitda_absolute_threshold']):
        factors.append(f"EV/EBITDA {ev_ebitda:.1f} is extremely high vs industry")
```

```
    return factors

def _check_quality_deterioration(self, historical: dict) -> List[str]:
    """Check for deteriorating business quality"""
    factors = []

    # ROE declining for 3 consecutive years
    roe_history = historical.get('roe_5_years', [])
    if len(roe_history) >= 3:
        declining_count = 0
        for i in range(1, len(roe_history)):
            if roe_history[i] < roe_history[i-1]:
                declining_count += 1
            else:
                declining_count = 0  # Reset counter

            if declining_count >= self.HIGH_RISK_THRESHOLDS['roe_declining_years']:
                factors.append("ROE declining for 3+ consecutive years")
                break

    # Negative FCF for 2+ years
    fcf_history = historical.get('fcf_5_years', [])
    negative_years = sum(1 for fcf in fcf_history[-2:] if fcf and fcf < 0)
    if negative_years >= self.HIGH_RISK_THRESHOLDS['fcf_negative_years']:
        factors.append(f"Free cash flow negative for {negative_years} years")

    # Gross margin declining >20% over 3 years
    margin_history = historical.get('gross_margins_5_years', [])
    if len(margin_history) >= 3:
        margin_change = (margin_history[-1] - margin_history[-3]) / margin_history[-3]
        if margin_change < -self.HIGH_RISK_THRESHOLDS['margin_decline_threshold']:
            factors.append(f"Gross margin declined {abs(margin_change)*100:.1f}% over 3 yea

    return factors

def _check_moderate_risks(self, data: dict, industry: dict) -> List[str]:
    """Check for moderate risk factors (🟠 WARNING)"""
    factors = []

    z_min, z_max = self.WARNING_THRESHOLDS['altman_z_score_range']
    if z_min <= data.get('altman_z_score', 999) < z_max:
        factors.append("Altman Z-Score in moderate risk zone")
```

```python
        ic_min, ic_max = self.WARNING_THRESHOLDS['interest_coverage_range']
        if ic_min <= data.get('interest_coverage', 999) < ic_max:
            factors.append("Interest coverage below optimal level")

        de_min, de_max = self.WARNING_THRESHOLDS['debt_to_equity_range']
        if de_min <= data.get('debt_to_equity', 0) <= de_max:
            factors.append("Debt levels elevated but manageable")

        # P/E moderately high vs industry
        pe_ratio = data.get('pe_ratio')
        industry_pe = industry.get('median_pe_ratio')
        if (pe_ratio and industry_pe and
            pe_ratio > industry_pe * self.WARNING_THRESHOLDS['pe_vs_industry_multiple']):
            factors.append(f"P/E {pe_ratio/industry_pe:.1f}x above industry median")

        return factors

    def _check_mild_concerns(self, data: dict, historical: dict) -> List[str]:
        """Check for mild concerns (🟡 CAUTION)"""
        factors = []

        z_min, z_max = self.CAUTION_THRESHOLDS['altman_z_score_range']
        if z_min <= data.get('altman_z_score', 999) <= z_max:
            factors.append("Altman Z-Score at lower end of safe zone")

        # Recent earnings volatility
        earnings_history = historical.get('earnings_5_years', [])
        if len(earnings_history) >= 3:
            recent_volatility = statistics.stdev(earnings_history[-3:])
            avg_earnings = statistics.mean(earnings_history[-3:])
            if avg_earnings > 0 and recent_volatility / avg_earnings > 0.3:
                factors.append("Recent earnings showing increased volatility")

        return factors

# Integration with investor scoring
def apply_risk_adjustment_to_score(base_score: float, risk_warning: dict) -> float:
    """
    Apply risk-based adjustments to investor scores
    """
    if risk_warning['warning_level'] == 'high_risk':
        return base_score * 0.7  # 30% penalty
    elif risk_warning['warning_level'] == 'warning':
        return base_score * 0.85  # 15% penalty
```

```
    elif risk_warning['warning_level'] == 'caution':
        return base_score * 0.95  # 5% penalty
    else:
        return base_score  # No adjustment
```

BACK-011B: Growth Calculations Engine

Type: Backend Task
Priority: Medium
Story Points: 8
Assignee: Backend Developer

Description: Implement exact growth calculation methodologies per professor specifications with proper handling of negative values.

Acceptance Criteria:

- [ ] 3-year CAGR calculations for revenue, earnings, FCF
- [ ] Proper handling of negative earnings periods
- [ ] Growth quality assessment
- [ ] Minimum data requirements enforcement
- [ ] Growth score calculation (0-100)

EXACT GROWTH FORMULAS (Per Professor):

python

```python
class GrowthCalculator:

    def calculate_revenue_growth_3yr(self, revenue_current: float, revenue_3yr_ago: float) -> O
        """
        3-Year Revenue CAGR = (Revenue_Current / Revenue_3_Years_Ago)^(1/3) - 1
        - Use TTM revenue for current
        - Minimum 3 years history required
        - Cap at ±100% for display
        """
        if not revenue_current or not revenue_3yr_ago or revenue_3yr_ago <= 0:
            return None

        try:
            cagr = (revenue_current / revenue_3yr_ago) ** (1/3) - 1
            # Cap at ±100% for display
            return max(min(cagr, 1.0), -1.0)
        except (ZeroDivisionError, ValueError):
            return None

    def calculate_earnings_growth_3yr(self, earnings_history: List[float]) -> Optional[float]:
        """
        3-Year Earnings CAGR = (EPS_Current / EPS_3_Years_Ago)^(1/3) - 1
        - Handle negative earnings: Use revenue growth if any year negative
        - Normalize for one-time charges where possible
        """
        if len(earnings_history) < 4:  # Need 4 data points for 3-year calculation
            return None

        eps_current = earnings_history[-1]
        eps_3yr_ago = earnings_history[-4]

        # Check for negative earnings in the period
        has_negative = any(eps <= 0 for eps in earnings_history[-4:])

        if has_negative:
            # Fall back to revenue growth - handled separately
            return None  # Caller should use revenue growth instead

        if eps_3yr_ago <= 0:
            return None

        try:
            cagr = (eps_current / eps_3yr_ago) ** (1/3) - 1
```

```
            return max(min(cagr, 1.0), -1.0)  # Cap at ±100%
        except (ZeroDivisionError, ValueError):
            return None

    def calculate_fcf_growth_3yr(self, fcf_current: float, fcf_3yr_ago: float) -> Optional[floa
        """
        3-Year FCF CAGR = (FCF_Current / FCF_3_Years_Ago)^(1/3) - 1
        - Most reliable growth metric
        - Use for companies with volatile earnings
        """
        if not fcf_current or not fcf_3yr_ago:
            return None

        # Handle negative FCF
        if fcf_3yr_ago <= 0:
            if fcf_current > 0:
                return 1.0  # Max positive growth (was negative, now positive)
            else:
                return None  # Both negative, can't calculate meaningful CAGR

        try:
            cagr = (fcf_current / fcf_3yr_ago) ** (1/3) - 1
            return max(min(cagr, 1.0), -1.0)
        except (ZeroDivisionError, ValueError):
            return None

    def calculate_growth_score(self, ticker: str) -> dict:
        """
        Calculate growth component score (0-100) for investor profiles
        """
        financial_data = self.get_financial_history(ticker, years=5)

        # Get growth rates
        revenue_growth = self.calculate_revenue_growth_3yr(
            financial_data['revenue'][-1],
            financial_data['revenue'][-4]
        )

        earnings_growth = self.calculate_earnings_growth_3yr(financial_data['earnings'])

        fcf_growth = self.calculate_fcf_growth_3yr(
            financial_data['fcf'][-1],
            financial_data['fcf'][-4]
        )
```

```
# Score each growth metric
revenue_score = self._score_growth_rate(revenue_growth, metric_type='revenue')
earnings_score = self._score_growth_rate(earnings_growth, metric_type='earnings')
fcf_score = self._score_growth_rate(fcf_growth, metric_type='fcf')

# Weighted combination
# FCF growth most reliable (40%), Revenue (35%), Earnings (25%)
available_scores = []
weights = []

if fcf_score is not None:
    available_scores.append(fcf_score)
    weights.append(0.40)
if revenue_score is not None:
    available_scores.append(revenue_score)
    weights.append(0.35)
if earnings_score is not None:
    available_scores.append(earnings_score)
    weights.append(0.25)

if not available_scores:
    total_score = 0
else:
    # Normalize weights
    total_weight = sum(weights)
    normalized_weights = [w / total_weight for w in weights]

    total_score = sum(
        score * weight
        for score, weight in zip(available_scores, normalized_weights)
    )

return {
    'total_score': round(total_score, 1),
    'components': {
        'revenue_growth_3yr': revenue_growth,
        'earnings_growth_3yr': earnings_growth,
        'fcf_growth_3yr': fcf_growth,
        'revenue_score': revenue_score,
        'earnings_score': earnings_score,
        'fcf_score': fcf_score
    },
    'data_quality': self._assess_growth_data_quality(financial_data)
```

```
    }

def _score_growth_rate(self, growth_rate: Optional[float], metric_type: str) -> Optional[ir
    """
    Score growth rate 0-100 based on metric type and investor expectations
    """
    if growth_rate is None:
        return None

    # Convert to percentage for easier comparison
    growth_pct = growth_rate * 100

    if metric_type == 'revenue':
        # Revenue growth scoring
        if growth_pct >= 20:
            return 100  # Excellent growth
        elif growth_pct >= 10:
            return 80   # Strong growth
        elif growth_pct >= 5:
            return 60   # Moderate growth
        elif growth_pct >= 0:
            return 40   # Slow growth
        elif growth_pct >= -5:
            return 20   # Slight decline
        else:
            return 0    # Significant decline

    elif metric_type == 'earnings':
        # Earnings growth scoring (higher expectations)
        if growth_pct >= 25:
            return 100
        elif growth_pct >= 15:
            return 80
        elif growth_pct >= 8:
            return 60
        elif growth_pct >= 0:
            return 40
        elif growth_pct >= -10:
            return 20
        else:
            return 0

    elif metric_type == 'fcf':
        # FCF growth scoring (most conservative)
```

```
            if growth_pct >= 15:
                return 100
            elif growth_pct >= 8:
                return 80
            elif growth_pct >= 3:
                return 60
            elif growth_pct >= 0:
                return 50   # FCF maintenance is valuable
            elif growth_pct >= -10:
                return 25
            else:
                return 0

        return 50  # Default neutral score

    def _assess_growth_data_quality(self, financial_data: dict) -> dict:
        """
        Assess quality and reliability of growth data
        """
        data_points = len(financial_data.get('revenue', []))

        quality_score = 0
        quality_notes = []

        # Data completeness (40 points)
        if data_points >= 5:
            quality_score += 40
        elif data_points >= 3:
            quality_score += 25
            quality_notes.append("Limited historical data (3-4 years)")
        else:
            quality_score += 0
            quality_notes.append("Insufficient data for reliable growth calculation")

        # Consistency (30 points)
        revenue_data = financial_data.get('revenue', [])
        if len(revenue_data) >= 3:
            # Check for unusual spikes or drops
            year_over_year_changes = []
            for i in range(1, len(revenue_data)):
                if revenue_data[i-1] > 0:
                    change = (revenue_data[i] - revenue_data[i-1]) / revenue_data[i-1]
                    year_over_year_changes.append(abs(change))
```

```
            if year_over_year_changes:
                avg_volatility = sum(year_over_year_changes) / len(year_over_year_changes)
                if avg_volatility < 0.2:  # <20% average change
                    quality_score += 30
                elif avg_volatility < 0.4:  # <40% average change
                    quality_score += 20
                    quality_notes.append("Moderate revenue volatility")
                else:
                    quality_score += 0
                    quality_notes.append("High revenue volatility affects growth reliability")

        # Trend clarity (30 points)
        if len(revenue_data) >= 4:
            # Simple trend analysis
            first_half = sum(revenue_data[:2]) / 2
            second_half = sum(revenue_data[-2:]) / 2

            if second_half > first_half * 1.1:  # Clear upward trend
                quality_score += 30
            elif second_half > first_half * 0.9:  # Stable
                quality_score += 20
            else:  # Declining trend
                quality_score += 10
                quality_notes.append("Declining revenue trend")

        return {
            'quality_score': quality_score,
            'quality_grade': 'High' if quality_score >= 80 else 'Medium' if quality_score >= 50
            'notes': quality_notes,
            'data_points_available': data_points
        }
```

BACK-012: Special Sector Handling Implementation

Type: Backend Task

Priority: Medium

Story Points: 13

Assignee: Backend Developer

Description: Implement special calculation logic for Financials, REITs, and Utilities per professor's exact specifications.

Acceptance Criteria:

- [ ] Financial sector custom ratios and scoring
- [ ] REIT-specific metrics (FFO, dividend yield)
- [ ] Utility sector regulatory considerations
- [ ] Sector-specific risk thresholds
- [ ] Modified Altman Z-Score for each sector

EXACT SECTOR SPECIFICATIONS (Per Professor):

1. FINANCIALS (Banks, Insurance):

python

```python
class FinancialSectorHandler:

    FINANCIAL_SECTORS = ['Banks', 'Insurance', 'Financial Services', 'Capital Markets']

    def is_financial_company(self, company_data: dict) -> bool:
        """Check if company is in financial sector"""
        return company_data.get('sector') in self.FINANCIAL_SECTORS

    def calculate_financial_ratios(self, company_data: dict) -> dict:
        """
        Key Ratios for Financials:
        - P/B instead of P/E (primary valuation)
        - ROE target: >12%
        - Tier 1 Capital Ratio: >10%
        - NIM (Net Interest Margin): Industry-relative
        - Loan Loss Provisions: <2% of loans
        """
        ratios = {}

        # P/B as primary valuation (not P/E)
        if company_data.get('market_cap') and company_data.get('book_value'):
            ratios['pb_ratio'] = company_data['market_cap'] / company_data['book_value']
            ratios['primary_valuation_metric'] = 'pb_ratio'

        # ROE (critical for banks)
        if company_data.get('net_income') and company_data.get('shareholders_equity'):
            ratios['roe'] = (company_data['net_income'] / company_data['shareholders_equity'])
            ratios['roe_quality'] = 'Excellent' if ratios['roe'] > 15 else 'Good' if ratios['ro

        # Tier 1 Capital Ratio (for banks)
        if company_data.get('tier_1_capital') and company_data.get('risk_weighted_assets'):
            ratios['tier_1_capital_ratio'] = (company_data['tier_1_capital'] / company_data['ri
            ratios['capital_adequacy'] = 'Strong' if ratios['tier_1_capital_ratio'] > 12 else '

        # Net Interest Margin
        if company_data.get('net_interest_income') and company_data.get('average_earning_assets
            ratios['net_interest_margin'] = (company_data['net_interest_income'] / company_data

        # Loan Loss Provisions
        if company_data.get('loan_loss_provisions') and company_data.get('total_loans'):
            ratios['loan_loss_ratio'] = (company_data['loan_loss_provisions'] / company_data['t
            ratios['credit_quality'] = 'Excellent' if ratios['loan_loss_ratio'] < 1 else 'Good'
```

```
        return ratios

    def calculate_financial_score(self, company_data: dict, ratios: dict) -> dict:
        """
        Financial sector scoring with different weights
        """
        score_components = {}

        # P/B Valuation (30% weight)
        pb_ratio = ratios.get('pb_ratio')
        if pb_ratio:
            if pb_ratio < 0.8:
                score_components['valuation'] = 100
            elif pb_ratio < 1.0:
                score_components['valuation'] = 80
            elif pb_ratio < 1.5:
                score_components['valuation'] = 60
            elif pb_ratio < 2.0:
                score_components['valuation'] = 40
            else:
                score_components['valuation'] = 20
        else:
            score_components['valuation'] = 50

        # ROE Quality (25% weight)
        roe = ratios.get('roe', 0)
        if roe > 15:
            score_components['profitability'] = 100
        elif roe > 12:
            score_components['profitability'] = 80
        elif roe > 8:
            score_components['profitability'] = 60
        elif roe > 5:
            score_components['profitability'] = 40
        else:
            score_components['profitability'] = 20

        # Capital Adequacy (25% weight)
        tier_1_ratio = ratios.get('tier_1_capital_ratio', 0)
        if tier_1_ratio > 12:
            score_components['financial_health'] = 100
        elif tier_1_ratio > 10:
            score_components['financial_health'] = 80
        elif tier_1_ratio > 8:
```

```
            score_components['financial_health'] = 60
        else:
            score_components['financial_health'] = 20

        # Credit Quality (20% weight)
        loan_loss_ratio = ratios.get('loan_loss_ratio', 0)
        if loan_loss_ratio < 1:
            score_components['quality'] = 100
        elif loan_loss_ratio < 2:
            score_components['quality'] = 80
        elif loan_loss_ratio < 3:
            score_components['quality'] = 60
        else:
            score_components['quality'] = 20

        # Calculate weighted score
        weights = {'valuation': 0.30, 'profitability': 0.25, 'financial_health': 0.25, 'quality
        total_score = sum(score_components[key] * weights[key] for key in weights.keys())

        return {
            'total_score': total_score,
            'components': score_components,
            'sector_specific_notes': [
                "Traditional debt ratios excluded (different capital structure)",
                "Standard Altman Z-Score not applicable",
                "P/B ratio used as primary valuation metric"
            ]
        }

    def get_financial_exclusions(self) -> List[str]:
        """
        Ratios to exclude for financial companies
        """
        return [
            'debt_to_equity',  # Not meaningful for banks
            'current_ratio',   # Different working capital structure
            'altman_z_score',  # Use bank-specific model
            'pe_ratio'         # Secondary to P/B for banks
        ]
```

2. REITs (Real Estate Investment Trusts):

python

```python
class REITSectorHandler:

    REIT_INDICATORS = ['REIT', 'Real Estate Investment Trust', 'Real Estate']

    def is_reit_company(self, company_data: dict) -> bool:
        """Check if company is a REIT"""
        industry = company_data.get('industry', '').lower()
        company_name = company_data.get('company_name', '').lower()
        return any(indicator.lower() in industry or indicator.lower() in company_name
                   for indicator in self.REIT_INDICATORS)

    def calculate_reit_ratios(self, company_data: dict) -> dict:
        """
        Key Metrics for REITs:
        - P/FFO (Price to Funds From Operations) instead of P/E
        - Dividend Yield: >4% typically expected
        - Debt/Total Assets: <50%
        - Occupancy Rates: >90%
        - FFO Growth Rate: 3-7% annually
        """
        ratios = {}

        # P/FFO (primary valuation metric)
        if company_data.get('current_price') and company_data.get('ffo_per_share'):
            ratios['p_ffo_ratio'] = company_data['current_price'] / company_data['ffo_per_share
            ratios['primary_valuation_metric'] = 'p_ffo_ratio'

        # Dividend Yield
        if company_data.get('annual_dividend') and company_data.get('current_price'):
            ratios['dividend_yield'] = (company_data['annual_dividend'] / company_data['current
            ratios['dividend_quality'] = 'Excellent' if ratios['dividend_yield'] > 6 else 'Good

        # Debt to Total Assets (not equity)
        if company_data.get('total_debt') and company_data.get('total_assets'):
            ratios['debt_to_assets'] = (company_data['total_debt'] / company_data['total_assets
            ratios['leverage_quality'] = 'Conservative' if ratios['debt_to_assets'] < 35 else '

        # Occupancy Rate
        if company_data.get('occupied_square_feet') and company_data.get('total_square_feet'):
            ratios['occupancy_rate'] = (company_data['occupied_square_feet'] / company_data['to
            ratios['occupancy_quality'] = 'Excellent' if ratios['occupancy_rate'] > 95 else 'Go

        # FFO Growth (3-year)
```

```
    if company_data.get('ffo_3yr_cagr'):
        ratios['ffo_growth_3yr'] = company_data['ffo_3yr_cagr'] * 100
        ratios['growth_quality'] = 'Excellent' if ratios['ffo_growth_3yr'] > 7 else 'Good'

    return ratios

def calculate_reit_score(self, company_data: dict, ratios: dict) -> dict:
    """
    REIT-specific scoring methodology
    """
    score_components = {}

    # P/FFO Valuation (25% weight)
    p_ffo = ratios.get('p_ffo_ratio')
    if p_ffo:
        if p_ffo < 12:
            score_components['valuation'] = 100
        elif p_ffo < 15:
            score_components['valuation'] = 80
        elif p_ffo < 20:
            score_components['valuation'] = 60
        elif p_ffo < 25:
            score_components['valuation'] = 40
        else:
            score_components['valuation'] = 20
    else:
        score_components['valuation'] = 50

    # Dividend Yield (25% weight)
    div_yield = ratios.get('dividend_yield', 0)
    if div_yield > 6:
        score_components['income'] = 100
    elif div_yield > 4:
        score_components['income'] = 80
    elif div_yield > 3:
        score_components['income'] = 60
    elif div_yield > 2:
        score_components['income'] = 40
    else:
        score_components['income'] = 20

    # Financial Health (20% weight)
    debt_to_assets = ratios.get('debt_to_assets', 0)
    if debt_to_assets < 35:
```

```
        score_components['financial_health'] = 100
    elif debt_to_assets < 45:
        score_components['financial_health'] = 80
    elif debt_to_assets < 55:
        score_components['financial_health'] = 60
    else:
        score_components['financial_health'] = 20

    # Operational Quality (15% weight)
    occupancy = ratios.get('occupancy_rate', 0)
    if occupancy > 95:
        score_components['quality'] = 100
    elif occupancy > 90:
        score_components['quality'] = 80
    elif occupancy > 85:
        score_components['quality'] = 60
    else:
        score_components['quality'] = 20

    # Growth (15% weight)
    ffo_growth = ratios.get('ffo_growth_3yr', 0)
    if ffo_growth > 7:
        score_components['growth'] = 100
    elif ffo_growth > 3:
        score_components['growth'] = 80
    elif ffo_growth > 0:
        score_components['growth'] = 60
    else:
        score_components['growth'] = 20

    # Calculate weighted score
    weights = {'valuation': 0.25, 'income': 0.25, 'financial_health': 0.20, 'quality': 0.15
    total_score = sum(score_components[key] * weights[key] for key in weights.keys())

    return {
        'total_score': total_score,
        'components': score_components,
        'sector_specific_notes': [
            "High dividend payout ratios are normal for REITs",
            "Focus on property quality and location",
            "FFO more relevant than traditional earnings"
        ]
    }
```

3. UTILITIES:

python

```python
class UtilitySectorHandler:

    UTILITY_INDICATORS = ['Utilities', 'Electric', 'Gas', 'Water', 'Energy Distribution']

    def is_utility_company(self, company_data: dict) -> bool:
        """Check if company is a utility"""
        sector = company_data.get('sector', '').lower()
        industry = company_data.get('industry', '').lower()
        return any(indicator.lower() in sector or indicator.lower() in industry
                   for indicator in self.UTILITY_INDICATORS)

    def calculate_utility_ratios(self, company_data: dict) -> dict:
        """
        Key Characteristics for Utilities:
        - Regulated ROE: 8-12% typical
        - Dividend Yield: 3-6%
        - Interest Coverage: >2.5x minimum
        - Rate Base Growth: 2-5% annually
        - Regulatory Environment: Critical factor
        """
        ratios = {}

        # Regulated ROE
        if company_data.get('net_income') and company_data.get('shareholders_equity'):
            ratios['roe'] = (company_data['net_income'] / company_data['shareholders_equity'])
            ratios['roe_quality'] = 'Excellent' if 10 <= ratios['roe'] <= 14 else 'Good' if 8 <

        # Dividend Yield and Coverage
        if company_data.get('annual_dividend') and company_data.get('current_price'):
            ratios['dividend_yield'] = (company_data['annual_dividend'] / company_data['current
            ratios['dividend_quality'] = 'Excellent' if 4 <= ratios['dividend_yield'] <= 6 else

        # Interest Coverage (critical for utilities with high debt)
        if company_data.get('ebit') and company_data.get('interest_expense'):
            ratios['interest_coverage'] = company_data['ebit'] / company_data['interest_expense
            ratios['debt_safety'] = 'Strong' if ratios['interest_coverage'] > 4 else 'Adequate'

        # Rate Base Growth
        if company_data.get('rate_base_3yr_cagr'):
            ratios['rate_base_growth'] = company_data['rate_base_3yr_cagr'] * 100
            ratios['growth_quality'] = 'Excellent' if 4 <= ratios['rate_base_growth'] <= 6 else

        return ratios
```

```python
def calculate_utility_score(self, company_data: dict, ratios: dict) -> dict:
    """
    Utility-specific scoring with emphasis on stability
    """
    score_components = {}

    # Dividend Sustainability (30% weight - critical for utility investors)
    div_yield = ratios.get('dividend_yield', 0)
    interest_coverage = ratios.get('interest_coverage', 0)

    dividend_score = 0
    if 4 <= div_yield <= 6 and interest_coverage > 3:
        dividend_score = 100
    elif 3 <= div_yield <= 7 and interest_coverage > 2.5:
        dividend_score = 80
    elif div_yield > 0 and interest_coverage > 2:
        dividend_score = 60
    else:
        dividend_score = 20
    score_components['dividend_sustainability'] = dividend_score

    # Regulatory ROE (25% weight)
    roe = ratios.get('roe', 0)
    if 10 <= roe <= 14:
        score_components['profitability'] = 100  # Optimal regulated range
    elif 8 <= roe <= 16:
        score_components['profitability'] = 80
    elif 6 <= roe <= 18:
        score_components['profitability'] = 60
    else:
        score_components['profitability'] = 40

    # Financial Stability (20% weight)
    if interest_coverage > 4:
        score_components['financial_health'] = 100
    elif interest_coverage > 3:
        score_components['financial_health'] = 80
    elif interest_coverage > 2.5:
        score_components['financial_health'] = 60
    else:
        score_components['financial_health'] = 20

    # Growth (15% weight - lower expectations)
```

```
        rate_base_growth = ratios.get('rate_base_growth', 0)
        if 4 <= rate_base_growth <= 6:
            score_components['growth'] = 100
        elif 2 <= rate_base_growth <= 8:
            score_components['growth'] = 80
        elif rate_base_growth > 0:
            score_components['growth'] = 60
        else:
            score_components['growth'] = 40

        # Regulatory Environment (10% weight)
        # This would require additional data about state regulations
        score_components['regulatory'] = 75  # Default neutral score

        # Calculate weighted score
        weights = {
            'dividend_sustainability': 0.30,
            'profitability': 0.25,
            'financial_health': 0.20,
            'growth': 0.15,
            'regulatory': 0.10
        }
        total_score = sum(score_components[key] * weights[key] for key in weights.keys())

        return {
            'total_score': total_score,
            'components': score_components,
            'sector_specific_notes': [
                "Lower growth expectations for regulated utilities",
                "Emphasis on dividend sustainability",
                "Regulatory risk assessment crucial",
                "Interest coverage critical due to high leverage"
            ]
        }
```

Integration with Main Scoring System:

python

```python
class SectorAwareScoring:

    def __init__(self):
        self.financial_handler = FinancialSectorHandler()
        self.reit_handler = REITSectorHandler()
        self.utility_handler = UtilitySectorHandler()

    def calculate_sector_adjusted_score(self, ticker: str, investor_type: str) -> dict:
        """
        Main entry point for sector-aware scoring
        """
        company_data = self.get_company_data(ticker)

        # Determine sector and use appropriate handler
        if self.financial_handler.is_financial_company(company_data):
            return self._calculate_financial_score(company_data, investor_type)
        elif self.reit_handler.is_reit_company(company_data):
            return self._calculate_reit_score(company_data, investor_type)
        elif self.utility_handler.is_utility_company(company_data):
            return self._calculate_utility_score(company_data, investor_type)
        else:
            # Use standard scoring methodology
            return self._calculate_standard_score(company_data, investor_type)

    def get_sector_exclusions(self, company_data: dict) -> List[str]:
        """
        Get list of ratios to exclude based on sector
        """
        if self.financial_handler.is_financial_company(company_data):
            return self.financial_handler.get_financial_exclusions()
        elif self.reit_handler.is_reit_company(company_data):
            return ['pe_ratio', 'eps_growth']  # Use FFO metrics instead
        elif self.utility_handler.is_utility_company(company_data):
            return []  # Standard ratios apply but with different thresholds
        else:
            return []
```

BACK-013: CRON Job Orchestrator with Smart Scheduling

Type: Backend Task

Priority: High

Story Points: 13

Assignee: Senior Backend Developer

Description: Implement daily cron jobs with professor's rate-limit strategy and intelligent error handling.

Acceptance Criteria:

- [ ] Monday-Friday different update patterns per professor's plan
- [ ] Graceful handling of API failures with cascading fallbacks
- [ ] Progress tracking and resumption capability
- [ ] Email alerts for critical failures
- [ ] Performance monitoring and optimization
- [ ] Priority-based company selection

PROFESSOR'S EXACT SCHEDULING STRATEGY:

Rate Limit Distribution (1500 companies across 5 weekdays):

```bash
# Monday - Yahoo Finance heavy (400 companies)
0 2 * * 1 python update_fundamentals.py --provider yahoo --limit 400 --priority high

# Tuesday - Finnhub focus (300 companies)
0 2 * * 2 python update_fundamentals.py --provider finnhub --limit 300 --priority medium

# Wednesday - Yahoo continued (400 companies)
0 2 * * 3 python update_fundamentals.py --provider yahoo --limit 400 --priority medium

# Thursday - Catch-up day (300 companies, mixed providers)
0 2 * * 4 python update_fundamentals.py --provider mixed --limit 300 --priority low

# Friday - Alpha Vantage only (100 companies, rate limit 5/min)
0 2 * * 5 python update_fundamentals.py --provider alphavantage --limit 100 --priority lowest

# Sunday - Industry benchmarks calculation (weekly)
0 1 * * 0 python calculate_industry_benchmarks.py

# Daily - Investor scores recalculation (for updated companies only)
0 4 * * * python calculate_investor_scores.py --updated-only

# Weekly - Peer selection refresh
0 3 * * 0 python refresh_peer_selections.py
```

Python Implementation:

python

```python
class CronOrchestrator:

    def __init__(self):
        self.api_limiter = APIRateLimiter()
        self.company_prioritizer = CompanyPrioritizer()
        self.error_handler = ErrorHandler()

    def update_fundamentals_cron(self, provider: str, limit: int, priority: str):
        """
        Main cron entry point for fundamental data updates
        """
        try:
            # Get prioritized company list
            companies = self.company_prioritizer.get_companies_by_priority(
                limit=limit,
                priority_level=priority,
                provider_preference=provider
            )

            logger.info(f"Starting {provider} update for {len(companies)} companies")

            # Execute updates with progress tracking
            results = self.execute_batch_updates(companies, provider)

            # Log results and handle failures
            self.process_batch_results(results, provider)

            # Update industry benchmarks if enough companies updated
            if results['success_count'] > limit * 0.8:  # 80% success rate
                self.trigger_benchmark_update()

        except Exception as e:
            self.error_handler.handle_critical_error(e, f"{provider}_cron_failure")
            raise

    def execute_batch_updates(self, companies: List[str], provider: str) -> dict:
        """
        Execute updates with rate limiting and error handling
        """
        results = {
            'success_count': 0,
            'failure_count': 0,
            'rate_limited_count': 0,
```

```
        'failed_companies': [],
        'processing_time': 0
    }

    start_time = time.time()

    for i, ticker in enumerate(companies):
        try:
            # Check rate limits before each call
            if not self.api_limiter.can_make_request(provider):
                wait_time = self.api_limiter.get_wait_time(provider)
                if wait_time > 300:  # More than 5 minutes
                    logger.warning(f"Rate limit exceeded for {provider}, stopping batch")
                    results['rate_limited_count'] = len(companies) - i
                    break
                else:
                    time.sleep(wait_time)

            # Update company fundamentals
            success = self.update_single_company(ticker, provider)

            if success:
                results['success_count'] += 1
                # Update company priority and last update time
                self.company_prioritizer.mark_updated(ticker)
            else:
                results['failure_count'] += 1
                results['failed_companies'].append(ticker)

            # Progress logging every 50 companies
            if (i + 1) % 50 == 0:
                progress = (i + 1) / len(companies) * 100
                logger.info(f"Progress: {progress:.1f}% ({i + 1}/{len(companies)})")

        except Exception as e:
            logger.error(f"Error updating {ticker}: {str(e)}")
            results['failure_count'] += 1
            results['failed_companies'].append(ticker)

    results['processing_time'] = time.time() - start_time
    return results

def update_single_company(self, ticker: str, provider: str) -> bool:
    """
```

```
    Update single company with fallback providers
    """
    providers = self.get_provider_fallback_chain(provider)

    for attempt_provider in providers:
        try:
            # Check rate limits
            if not self.api_limiter.can_make_request(attempt_provider):
                continue

            # Fetch data
            fundamental_data = self.fetch_fundamental_data(ticker, attempt_provider)

            if self.validate_fundamental_data(fundamental_data):
                # Calculate ratios
                ratios = self.calculate_all_ratios(fundamental_data)

                # Store in database
                self.store_fundamental_data(ticker, fundamental_data, ratios, attempt_provi

                # Record successful API call
                self.api_limiter.record_successful_call(attempt_provider)

                return True

        except APIRateLimitError:
            logger.warning(f"Rate limit hit for {attempt_provider}")
            continue
        except APIError as e:
            logger.warning(f"API error for {ticker} with {attempt_provider}: {str(e)}")
            continue
        except Exception as e:
            logger.error(f"Unexpected error for {ticker} with {attempt_provider}: {str(e)}"
            continue

    # All providers failed
    logger.error(f"All providers failed for {ticker}")
    return False

def get_provider_fallback_chain(self, primary_provider: str) -> List[str]:
    """
    Get fallback provider chain per professor's specifications
    """
    if primary_provider == 'yahoo':
```

```
            return ['yahoo', 'finnhub', 'alphavantage']
        elif primary_provider == 'finnhub':
            return ['finnhub', 'yahoo', 'alphavantage']
        elif primary_provider == 'alphavantage':
            return ['alphavantage', 'yahoo', 'finnhub']
        elif primary_provider == 'mixed':
            return ['yahoo', 'finnhub', 'alphavantage']
        else:
            return ['yahoo', 'finnhub', 'alphavantage']

class CompanyPrioritizer:
    """
    Implement professor's exact priority algorithm
    """

    def get_companies_by_priority(self, limit: int, priority_level: str, provider_preference: s
        """
        Get companies using professor's priority algorithm:
        Priority 5: earnings ≤ 7 days
        Priority 4: no fundamental data
        Priority 3: data > 30 days old
        Priority 2: data > 90 days old
        Priority 1: recent data
        """

        query = """
        WITH company_priorities AS (
            SELECT
                s.ticker,
                s.company_name,
                s.next_earnings_date,
                s.fundamentals_last_update,
                s.data_priority as manual_priority,
                CASE
                    WHEN s.next_earnings_date <= CURRENT_DATE + INTERVAL '7 days' THEN 5
                    WHEN s.fundamentals_last_update IS NULL THEN 4
                    WHEN s.fundamentals_last_update < CURRENT_DATE - INTERVAL '30 days' THEN 3
                    WHEN s.fundamentals_last_update < CURRENT_DATE - INTERVAL '90 days' THEN 2
                    ELSE 1
                END as calculated_priority,
                s.market_cap
            FROM stocks s
            WHERE s.market_cap > 100000000  -- Min $100M market cap
        )
```

```
        SELECT ticker
        FROM company_priorities
        WHERE calculated_priority >= %s
        ORDER BY
            calculated_priority DESC,
            manual_priority DESC,
            market_cap DESC,
            fundamentals_last_update ASC NULLS FIRST
        LIMIT %s
        """

        # Map priority levels to minimum priority scores
        priority_mapping = {
            'high': 4,      # Only earnings soon + no data
            'medium': 3,    # Include 30+ day old data
            'low': 2,       # Include 90+ day old data
            'lowest': 1     # Include all companies
        }

        min_priority = priority_mapping.get(priority_level, 1)

        companies = self.db.execute_query(query, (min_priority, limit))
        return [company['ticker'] for company in companies]

    def mark_updated(self, ticker: str):
        """
        Update company's last update timestamp and adjust priority
        """
        query = """
        UPDATE stocks
        SET fundamentals_last_update = CURRENT_TIMESTAMP,
            data_priority = 1  -- Reset to low priority after update
        WHERE ticker = %s
        """
        self.db.execute_query(query, (ticker,))

class ErrorHandler:
    """
    Comprehensive error handling and alerting
    """

    def handle_critical_error(self, error: Exception, context: str):
        """
        Handle critical errors that require immediate attention
```

```
    """
    error_data = {
        'error_type': type(error).__name__,
        'error_message': str(error),
        'context': context,
        'timestamp': datetime.now(),
        'stack_trace': traceback.format_exc()
    }

    # Log to database
    self.log_error_to_db(error_data)

    # Send immediate alert for critical failures
    if self.is_critical_error(error, context):
        self.send_immediate_alert(error_data)

def process_batch_results(self, results: dict, provider: str):
    """
    Process batch results and send summary alerts
    """
    success_rate = results['success_count'] / (results['success_count'] + results['failure_

    # Log batch summary
    batch_summary = {
        'provider': provider,
        'success_count': results['success_count'],
        'failure_count': results['failure_count'],
        'success_rate': success_rate,
        'processing_time': results['processing_time'],
        'failed_companies': results['failed_companies'][:10]  # First 10 failures
    }

    self.log_batch_summary(batch_summary)

    # Alert if success rate too low
    if success_rate < 0.7:  # Less than 70% success
        self.send_low_success_rate_alert(batch_summary)

    # Schedule retries for failed companies
    if results['failed_companies']:
        self.schedule_retries(results['failed_companies'], provider)

def send_immediate_alert(self, error_data: dict):
    """
```

```
        Send immediate email alert for critical errors
        """
        subject = f"CRITICAL: {error_data['context']} Failed"
        body = f"""
        Critical error in fundamental data pipeline:

        Error: {error_data['error_type']}
        Message: {error_data['error_message']}
        Context: {error_data['context']}
        Time: {error_data['timestamp']}

        Stack Trace:
        {error_data['stack_trace']}

        Immediate action required.
        """

        self.email_service.send_alert(subject, body, priority='high')

    def schedule_retries(self, failed_companies: List[str], provider: str):
        """
        Schedule retry attempts for failed companies
        """
        for ticker in failed_companies:
            # Increase retry count
            retry_count = self.get_retry_count(ticker, provider)

            if retry_count < 3:  # Max 3 retries
                # Schedule retry with exponential backoff
                delay_hours = 2 ** retry_count  # 2, 4, 8 hours
                retry_time = datetime.now() + timedelta(hours=delay_hours)

                self.schedule_retry(ticker, provider, retry_time, retry_count + 1)
```

BACK-014: Data Quality & Validation Engine

Type: Backend Task

Priority: Medium

Story Points: 8

Assignee: Backend Developer

Description: Implement comprehensive data quality validation per professor's exact specifications.

Acceptance Criteria:

- [ ] Implement professor's data quality rules
- [ ] Confidence scoring system (0-100)
- [ ] Outlier detection and flagging
- [ ] Cross-validation between data sources
- [ ] Automatic correction for common issues

PROFESSOR'S EXACT DATA QUALITY SPECIFICATIONS:

python

```python
class DataQualityValidator:

    # Professor's validation rules
    VALIDATION_RULES = {
        'pe_ratio': {'min': -100, 'max': 1000, 'outlier_threshold': 3},  # 3 std dev
        'debt_to_equity': {'min': 0, 'max': 50, 'warning_threshold': 5},
        'roe': {'min': -100, 'max': 200, 'outlier_threshold': 2},  # 2 std dev
        'revenue_growth': {'min': -90, 'max': 1000, 'outlier_threshold': 3},
        'current_ratio': {'min': 0, 'max': 20, 'warning_threshold': 10},
        'gross_margin': {'min': -50, 'max': 100, 'outlier_threshold': 2}
    }

    def validate_fundamental_data(self, ticker: str, data: dict, source: str) -> dict:
        """
        Comprehensive validation per professor's specifications
        """
        validation_results = {
            'is_valid': True,
            'quality_score': 0,
            'confidence_level': 'High',
            'issues': [],
            'corrections_applied': [],
            'data_completeness': 0,
            'logical_consistency': True,
            'source_reliability': self.get_source_reliability(source)
        }

        # 1. Data Completeness Check (40% of quality score)
        completeness_score = self.check_data_completeness(data)
        validation_results['data_completeness'] = completeness_score

        # 2. Range Validation (30% of quality score)
        range_score = self.validate_data_ranges(data, validation_results)

        # 3. Logical Consistency (20% of quality score)
        consistency_score = self.check_logical_consistency(data, validation_results)

        # 4. Cross-Source Validation (10% of quality score)
        cross_validation_score = self.cross_validate_with_existing(ticker, data, source)

        # Calculate overall quality score
        validation_results['quality_score'] = (
            completeness_score * 0.40 +
```

```
            range_score * 0.30 +
            consistency_score * 0.20 +
            cross_validation_score * 0.10
        )

        # Determine confidence level
        validation_results['confidence_level'] = self.determine_confidence_level(
            validation_results['quality_score']
        )

        # Apply automatic corrections
        corrected_data = self.apply_automatic_corrections(data, validation_results)

        return validation_results, corrected_data

    def check_data_completeness(self, data: dict) -> float:
        """
        Check what percentage of required fields are available
        """
        required_fields = [
            'revenue', 'net_income', 'total_assets', 'shareholders_equity',
            'total_debt', 'current_assets', 'current_liabilities',
            'operating_income', 'cash_and_equivalents'
        ]

        available_count = sum(1 for field in required_fields if data.get(field) is not None)
        completeness_percentage = (available_count / len(required_fields)) * 100

        return completeness_percentage

    def validate_data_ranges(self, data: dict, validation_results: dict) -> float:
        """
        Validate that all ratios are within reasonable ranges
        """
        range_issues = 0
        total_checks = 0

        for metric, rules in self.VALIDATION_RULES.items():
            if metric in data and data[metric] is not None:
                value = data[metric]
                total_checks += 1

                # Check min/max bounds
                if value < rules['min'] or value > rules['max']:
```

```
                validation_results['issues'].append(
                    f"{metric} value {value} outside valid range [{rules['min']}, {rules['m
                )
                range_issues += 1

            # Check for outliers (if we have industry context)
            if 'outlier_threshold' in rules:
                industry_avg = self.get_industry_average(metric)
                industry_std = self.get_industry_std_dev(metric)

                if industry_avg and industry_std:
                    z_score = abs(value - industry_avg) / industry_std
                    if z_score > rules['outlier_threshold']:
                        validation_results['issues'].append(
                            f"{metric} is {z_score:.1f} standard deviations from industry a
                        )

    # Calculate range score
    if total_checks == 0:
        return 0

    range_score = max(0, (total_checks - range_issues) / total_checks * 100)
    return range_score

def check_logical_consistency(self, data: dict, validation_results: dict) -> float:
    """
    Check for logical relationships between financial metrics
    """
    consistency_issues = 0
    total_checks = 0

    # Check 1: Current Assets ≥ Cash (if both available)
    if data.get('current_assets') and data.get('cash_and_equivalents'):
        total_checks += 1
        if data['cash_and_equivalents'] > data['current_assets']:
            validation_results['issues'].append(
                "Cash exceeds current assets - logical inconsistency"
            )
            consistency_issues += 1

    # Check 2: Total Assets ≥ Current Assets
    if data.get('total_assets') and data.get('current_assets'):
        total_checks += 1
        if data['current_assets'] > data['total_assets']:
```

```
                validation_results['issues'].append(
                    "Current assets exceed total assets - logical inconsistency"
                )
                consistency_issues += 1

        # Check 3: Revenue ≥ Net Income (for profitable companies)
        if data.get('revenue') and data.get('net_income'):
            total_checks += 1
            if data['net_income'] > data['revenue'] and data['net_income'] > 0:
                validation_results['issues'].append(
                    "Net income exceeds revenue - unusual for operating companies"
                )
                consistency_issues += 1

        # Check 4: Shareholders Equity = Total Assets - Total Liabilities (approximate)
        if all(data.get(field) for field in ['total_assets', 'total_debt', 'shareholders_equity
            total_checks += 1
            # Approximate total liabilities as total debt (simplified)
            implied_equity = data['total_assets'] - data['total_debt']
            equity_difference = abs(implied_equity - data['shareholders_equity'])

            # Allow 20% variance for simplification
            if equity_difference > data['shareholders_equity'] * 0.2:
                validation_results['issues'].append(
                    f"Balance sheet equation inconsistency: Assets - Debt ≠ Equity"
                )
                consistency_issues += 1

        # Calculate consistency score
        if total_checks == 0:
            return 100  # No checks possible, assume consistent

        consistency_score = max(0, (total_checks - consistency_issues) / total_checks * 100)
        return consistency_score

    def cross_validate_with_existing(self, ticker: str, new_data: dict, source: str) -> float:
        """
        Cross-validate new data with existing data from other sources
        """
        existing_data = self.get_recent_data_from_other_sources(ticker, source)

        if not existing_data:
            return 75  # Neutral score when no comparison data available
```

```
        cross_validation_score = 100
        comparison_count = 0

        # Compare key metrics with 15% tolerance
        key_metrics = ['revenue', 'net_income', 'total_assets', 'shareholders_equity']

        for metric in key_metrics:
            if metric in new_data and metric in existing_data:
                new_value = new_data[metric]
                existing_value = existing_data[metric]
                comparison_count += 1

                if existing_value != 0:
                    variance = abs(new_value - existing_value) / abs(existing_value)

                    if variance > 0.15:  # >15% difference
                        cross_validation_score -= 20  # Penalty for large variance
                        logger.warning(
                            f"Large variance in {metric} for {ticker}: "
                            f"{source}={new_value}, other={existing_value}"
                        )

        return max(0, cross_validation_score)

    def apply_automatic_corrections(self, data: dict, validation_results: dict) -> dict:
        """
        Apply automatic corrections for common data issues
        """
        corrected_data = data.copy()

        # Correction 1: Negative book value handling
        if corrected_data.get('shareholders_equity', 0) < 0:
            # Flag but don't auto-correct - this is meaningful information
            validation_results['corrections_applied'].append(
                "Negative shareholders equity flagged (not corrected)"
            )

        # Correction 2: Extreme P/E ratios
        if 'pe_ratio' in corrected_data:
            if corrected_data['pe_ratio'] > 1000:
                corrected_data['pe_ratio'] = None
                validation_results['corrections_applied'].append(
                    "Extreme P/E ratio removed (>1000)"
                )
```

```
        # Correction 3: Missing current ratio calculation
        if (not corrected_data.get('current_ratio') and
            corrected_data.get('current_assets') and
            corrected_data.get('current_liabilities')):

            if corrected_data['current_liabilities'] > 0:
                corrected_data['current_ratio'] = (
                    corrected_data['current_assets'] / corrected_data['current_liabilities']
                )
                validation_results['corrections_applied'].append(
                    "Current ratio calculated from components"
                )

        return corrected_data

    def determine_confidence_level(self, quality_score: float) -> str:
        """
        Determine confidence level based on quality score (per professor)
        """
        if quality_score >= 80:
            return 'High'
        elif quality_score >= 60:
            return 'Medium'
        else:
            return 'Low'

    def calculate_confidence_adjustment(self, quality_score: float) -> float:
        """
        Calculate score adjustment based on data confidence (per professor)
        """
        if quality_score >= 80:
            return 1.0      # No adjustment for high confidence
        elif quality_score >= 60:
            return 0.95     # 5% reduction for medium confidence
        else:
            return 0.85     # 15% reduction for low confidence

    def get_source_reliability(self, source: str) -> float:
        """
        Source reliability scores (professor's hierarchy)
        """
        reliability_scores = {
            'yahoo': 0.85,      # Primary source, good coverage
```

```
        'finnhub': 0.80,     # Secondary source, reliable
        'alphavantage': 0.75 # Tertiary source, limited calls
    }

    return reliability_scores.get(source, 0.70)
```

BACK-015: CSV Configuration Management with Formula Evaluation

Type: Backend Task
Priority: Medium
Story Points: 8
Assignee: Backend Developer

Description: Build system to parse and manage CSV configuration files for scoring logic with professor's dynamic threshold support.

Acceptance Criteria:

- [ ] Parse and validate CSV files with exact column structures
- [ ] Store in `csv_configurations` table with versioning
- [ ] Formula evaluation for dynamic thresholds (e.g., "industry_avg*0.8")
- [ ] API endpoint for CSV updates
- [ ] Rollback capability for bad configurations
- [ ] Support for all professor-specified CSV files

CSV FILES TO HANDLE (Professor-Specified):

python

```
CSV_CONFIGURATION_FILES = {
    'valuation_logic.csv': {
        'columns': ['Investor_Type', 'Metric_Name', 'Weight_Percentage', 'Good_Threshold',
                    'Fair_Threshold', 'Poor_Threshold', 'Calculation_Formula',
                    'Industry_Adjustment', 'Explanation_Template'],
        'validation_rules': {
            'Weight_Percentage': {'type': 'int', 'min': 0, 'max': 100},
            'Industry_Adjustment': {'type': 'bool'},
            'Calculation_Formula': {'type': 'formula'}
        }
    },
    'quality_scoring.csv': {
        'columns': ['Component_Name', 'Industry_Type', 'Excellent_Range', 'Good_Range',
                    'Average_Range', 'Poor_Range', 'Special_Calculation', 'Metric_Description'],
        'validation_rules': {
            'Industry_Type': {'enum': ['General', 'Financial', 'REIT', 'Utility']},
            'Special_Calculation': {'enum': ['standard', 'financial_adjusted', 'reit_specific']
        }
    },
    'investor_explanations.csv': {
        'columns': ['Investor_Type', 'Score_Range', 'Overall_Rating', 'Explanation_Template',
                    'Action_Suggestion'],
        'validation_rules': {
            'Investor_Type': {'enum': ['conservative', 'garp', 'deep_value']},
            'Score_Range': {'type': 'range', 'format': '0-19'},
            'Overall_Rating': {'enum': ['STRONG BUY', 'BUY', 'HOLD', 'SELL', 'STRONG SELL']}
        }
    },
    'educational_content.csv': {
        'columns': ['Metric_Name', 'Beginner_Explanation', 'Intermediate_Explanation',
                    'Advanced_Explanation', 'Why_It_Matters', 'Red_Flag_Levels', 'Learn_More_Lir
        'validation_rules': {
            'Red_Flag_Levels': {'type': 'threshold'},
            'Learn_More_Link': {'type': 'url'}
        }
    },
    'warning_thresholds.csv': {
        'columns': ['Warning_Level', 'Metric_Name', 'Condition_Logic', 'Warning_Text',
                    'Icon', 'Priority'],
        'validation_rules': {
            'Warning_Level': {'enum': ['caution', 'warning', 'high_risk']},
            'Priority': {'type': 'int', 'min': 1, 'max': 3},
            'Condition_Logic': {'type': 'formula'}
```

```
        }
    },
    'metric_descriptions.csv': {
        'columns': ['Metric_Name', 'Display_Name', 'Category', 'Format_Type',
                    'Units', 'Tooltip_Text', 'Calculation_Method'],
        'validation_rules': {
            'Category': {'enum': ['valuation', 'profitability', 'financial_health', 'growth']},
            'Format_Type': {'enum': ['percentage', 'ratio', 'currency', 'number']}
        }
    }
}
```

Implementation:

python

```python
class CSVConfigurationManager:

    def __init__(self):
        self.formula_evaluator = FormulaEvaluator()
        self.validator = CSVValidator()

    def load_csv_configuration(self, file_name: str, file_content: str, version: int) -> dict:
        """
        Load and validate CSV configuration file
        """
        try:
            # Parse CSV content
            parsed_data = self.parse_csv_content(file_content)

            # Validate structure and content
            validation_result = self.validator.validate_csv_structure(
                file_name, parsed_data, CSV_CONFIGURATION_FILES[file_name]
            )

            if not validation_result['is_valid']:
                raise ValueError(f"CSV validation failed: {validation_result['errors']}")

            # Test formula evaluation
            formula_test_result = self.test_formula_evaluation(file_name, parsed_data)

            # Store in database with versioning
            config_id = self.store_csv_configuration(file_name, parsed_data, version)

            # Activate new configuration
            self.activate_configuration(file_name, version)

            return {
                'success': True,
                'config_id': config_id,
                'validation_result': validation_result,
                'formula_test_result': formula_test_result
            }

        except Exception as e:
            logger.error(f"Failed to load CSV configuration {file_name}: {str(e)}")
            raise

    def evaluate_dynamic_threshold(self, formula: str, context: dict) -> float:
```

```
    """
    Evaluate dynamic thresholds like "industry_avg*0.8"

    Supported variables:
    - industry_avg: Industry average for the metric
    - industry_median: Industry median
    - peer_avg: Average of 3 peers
    - historical_avg: Company's 5-year average
    """
    try:
        # Replace variables with actual values
        evaluated_formula = formula

        for variable, value in context.items():
            if value is not None:
                evaluated_formula = evaluated_formula.replace(variable, str(value))

        # Safely evaluate mathematical expression
        result = self.formula_evaluator.safe_eval(evaluated_formula)
        return float(result)

    except Exception as e:
        logger.warning(f"Formula evaluation failed for '{formula}': {str(e)}")
        return None

def get_threshold_for_metric(self, metric_name: str, investor_type: str,
                             context: dict) -> dict:
    """
    Get dynamic thresholds for a metric based on current context
    """
    # Get configuration for metric
    config = self.get_active_configuration('valuation_logic.csv')

    metric_config = None
    for row in config['data']:
        if (row['Metric_Name'] == metric_name and
            row['Investor_Type'] == investor_type):
            metric_config = row
            break

    if not metric_config:
        return None

    # Evaluate thresholds
```

```
        thresholds = {}

        for threshold_type in ['Good_Threshold', 'Fair_Threshold', 'Poor_Threshold']:
            formula = metric_config[threshold_type]

            if self.is_formula(formula):
                thresholds[threshold_type] = self.evaluate_dynamic_threshold(formula, context)
            else:
                thresholds[threshold_type] = float(formula)

        return {
            'metric_name': metric_name,
            'investor_type': investor_type,
            'thresholds': thresholds,
            'explanation': metric_config['Explanation_Template'],
            'industry_adjustment': metric_config['Industry_Adjustment'] == 'true'
        }

    def is_formula(self, value: str) -> bool:
        """
        Check if string contains formula (has variables or operators)
        """
        formula_indicators = ['industry_avg', 'peer_avg', 'historical_avg', '*', '/', '+', '-']
        return any(indicator in value for indicator in formula_indicators)

class FormulaEvaluator:
    """
    Safe formula evaluation for dynamic thresholds
    """

    ALLOWED_OPERATORS = ['+', '-', '*', '/', '(', ')', '.']
    ALLOWED_FUNCTIONS = ['abs', 'min', 'max', 'round']

    def safe_eval(self, expression: str) -> float:
        """
        Safely evaluate mathematical expression
        """
        # Remove any non-allowed characters
        cleaned = self.clean_expression(expression)

        # Use restricted evaluation
        try:
            # Create safe namespace
            safe_dict = {
```

```
                '__builtins__': {},
                'abs': abs,
                'min': min,
                'max': max,
                'round': round
            }

            result = eval(cleaned, safe_dict)
            return float(result)

        except Exception as e:
            raise ValueError(f"Invalid formula: {expression}")

    def clean_expression(self, expression: str) -> str:
        """
        Clean and validate expression for safety
        """
        # Allow only numbers, operators, and parentheses
        allowed_chars = set('0123456789+-*/.() ')
        cleaned = ''.join(c for c in expression if c in allowed_chars)

        # Basic validation
        if not cleaned.strip():
            raise ValueError("Empty expression")

        return cleaned

class CSVValidator:
    """
    Validate CSV files against expected structure
    """

    def validate_csv_structure(self, file_name: str, data: List[dict],
                               expected_structure: dict) -> dict:
        """
        Validate CSV data against expected structure
        """
        validation_result = {
            'is_valid': True,
            'errors': [],
            'warnings': []
        }

        if not data:
```

```
        validation_result['is_valid'] = False
        validation_result['errors'].append("CSV file is empty")
        return validation_result

    # Check columns
    expected_columns = set(expected_structure['columns'])
    actual_columns = set(data[0].keys())

    missing_columns = expected_columns - actual_columns
    extra_columns = actual_columns - expected_columns

    if missing_columns:
        validation_result['is_valid'] = False
        validation_result['errors'].append(f"Missing columns: {missing_columns}")

    if extra_columns:
        validation_result['warnings'].append(f"Extra columns: {extra_columns}")

    # Validate data types and constraints
    validation_rules = expected_structure.get('validation_rules', {})

    for row_idx, row in enumerate(data):
        for column, rule in validation_rules.items():
            if column in row and row[column]:
                value = row[column]

                # Type validation
                if rule.get('type') == 'int':
                    try:
                        int_value = int(value)
                        if 'min' in rule and int_value < rule['min']:
                            validation_result['errors'].append(
                                f"Row {row_idx}: {column} value {int_value} below minimum {
                            )
                        if 'max' in rule and int_value > rule['max']:
                            validation_result['errors'].append(
                                f"Row {row_idx}: {column} value {int_value} above maximum {
                            )
                    except ValueError:
                        validation_result['errors'].append(
                            f"Row {row_idx}: {column} is not a valid integer"
                        )

                # Enum validation
```

```
                if 'enum' in rule and value not in rule['enum']:
                    validation_result['errors'].append(
                        f"Row {row_idx}: {column} value '{value}' not in allowed values {ru
                    )

                # URL validation
                if rule.get('type') == 'url' and not self.is_valid_url(value):
                    validation_result['warnings'].append(
                        f"Row {row_idx}: {column} may not be a valid URL"
                    )

        if validation_result['errors']:
            validation_result['is_valid'] = False

        return validation_result

    def is_valid_url(self, url: str) -> bool:
        """Basic URL validation"""
        return url.startswith(('http://', 'https://')) and '.' in url
```

BACK-016: Performance Optimization & Database Indexing

Type: Backend Task
Priority: Medium
Story Points: 8
Assignee: Backend Developer

Description: Optimize database queries and implement caching for frequent lookups per professor's performance requirements.

Acceptance Criteria:

- [ ] Query performance analysis for all major operations
- [ ] Add composite indexes for common query patterns
- [ ] Redis caching for industry benchmarks and peer data
- [ ] Query optimization with CTEs and materialized views
- [ ] Performance monitoring (<100ms requirement)

PERFORMANCE OPTIMIZATIONS:

sql

```sql
-- Key Optimizations per Professor Requirements

-- 1. Optimize latest ratios lookup (most frequent query)
CREATE INDEX CONCURRENTLY idx_ratios_latest_lookup
ON financial_ratios (ticker, calculation_date DESC)
WHERE calculation_date >= CURRENT_DATE - INTERVAL '90 days';

-- 2. Optimize industry benchmark queries
CREATE INDEX CONCURRENTLY idx_benchmarks_industry_date
ON industry_benchmarks (industry_code, calculation_date DESC);

-- 3. Optimize peer lookup queries
CREATE INDEX CONCURRENTLY idx_stocks_peer_lookup
ON stocks (industry, market_cap DESC)
WHERE market_cap > 100000000;

-- 4. Optimize priority-based company selection
CREATE INDEX CONCURRENTLY idx_stocks_priority_selection
ON stocks (data_priority DESC, fundamentals_last_update ASC NULLS FIRST, market_cap DESC)
WHERE market_cap > 100000000;

-- 5. Optimize earnings calendar queries
CREATE INDEX CONCURRENTLY idx_earnings_calendar_date_priority
ON earnings_calendar (earnings_date, priority_level DESC)
WHERE earnings_date >= CURRENT_DATE;

-- 6. Create materialized view for latest company metrics
CREATE MATERIALIZED VIEW mv_latest_company_metrics AS
SELECT DISTINCT ON (fr.ticker)
    fr.ticker,
    fr.calculation_date,
    fr.pe_ratio,
    fr.pb_ratio,
    fr.roe,
    fr.debt_to_equity,
    fr.altman_z_score,
    s.company_name,
    s.sector,
    s.industry,
    s.market_cap,
    s.peer_1_ticker,
    s.peer_2_ticker,
    s.peer_3_ticker
```

```
FROM financial_ratios fr
JOIN stocks s ON fr.ticker = s.ticker
ORDER BY fr.ticker, fr.calculation_date DESC;

-- Refresh materialized view daily
CREATE INDEX ON mv_latest_company_metrics (ticker);
CREATE INDEX ON mv_latest_company_metrics (sector, industry);

-- 7. Optimize investor score queries
CREATE INDEX CONCURRENTLY idx_investor_scores_latest
ON investor_scores (ticker, calculation_date DESC);
```

Python Caching Implementation:

python

```python
import redis
import json
from typing import Optional, Dict, Any
from datetime import timedelta

class PerformanceOptimizer:

    def __init__(self):
        self.redis_client = redis.Redis(host='localhost', port=6379, db=0)
        self.cache_ttl = {
            'industry_benchmarks': 86400,  # 24 hours
            'peer_data': 604800,           # 7 days
            'company_ratios': 3600,        # 1 hour
            'sector_mappings': 2592000     # 30 days
        }

    def get_cached_industry_benchmarks(self, industry: str) -> Optional[Dict]:
        """
        Get cached industry benchmarks
        """
        cache_key = f"industry_benchmarks:{industry}"
        cached_data = self.redis_client.get(cache_key)

        if cached_data:
            return json.loads(cached_data)

        # If not cached, fetch from database and cache
        benchmarks = self.fetch_industry_benchmarks_from_db(industry)
        if benchmarks:
            self.redis_client.setex(
                cache_key,
                self.cache_ttl['industry_benchmarks'],
                json.dumps(benchmarks)
            )

        return benchmarks

    def get_cached_peer_data(self, ticker: str) -> Optional[Dict]:
        """
        Get cached peer comparison data
        """
        cache_key = f"peer_data:{ticker}"
        cached_data = self.redis_client.get(cache_key)
```

```
    if cached_data:
        return json.loads(cached_data)

    # Fetch peer data and cache
    peer_data = self.fetch_peer_data_from_db(ticker)
    if peer_data:
        self.redis_client.setex(
            cache_key,
            self.cache_ttl['peer_data'],
            json.dumps(peer_data)
        )

    return peer_data

def optimize_bulk_ratio_calculation(self, tickers: List[str]) -> Dict[str, Dict]:
    """
    Optimized bulk calculation of ratios
    """
    # Use single query to fetch all needed data
    query = """
    SELECT
        ticker,
        revenue, net_income, total_assets, shareholders_equity,
        total_debt, current_assets, current_liabilities,
        operating_income, cash_and_equivalents, market_cap
    FROM mv_latest_company_metrics
    WHERE ticker = ANY(%s)
    """

    results = self.db.execute_query(query, (tickers,))

    # Calculate ratios in batch
    ratio_results = {}
    for row in results:
        ticker = row['ticker']
        ratio_results[ticker] = self.calculate_ratios_from_row(row)

    return ratio_results

def refresh_materialized_views(self):
    """
    Refresh materialized views (called by daily cron)
    """
```

```
        views_to_refresh = [
            'mv_latest_company_metrics'
        ]

        for view in views_to_refresh:
            start_time = time.time()
            self.db.execute_query(f"REFRESH MATERIALIZED VIEW {view}")
            refresh_time = time.time() - start_time

            logger.info(f"Refreshed {view} in {refresh_time:.2f} seconds")

    def monitor_query_performance(self):
        """
        Monitor and log slow queries
        """
        slow_query_threshold = 0.1  # 100ms as per professor requirement

        # Get slow queries from PostgreSQL logs
        query = """
        SELECT query, mean_exec_time, calls, total_exec_time
        FROM pg_stat_statements
        WHERE mean_exec_time > %s
        ORDER BY mean_exec_time DESC
        LIMIT 10
        """

        slow_queries = self.db.execute_query(query, (slow_query_threshold * 1000,))

        if slow_queries:
            logger.warning(f"Found {len(slow_queries)} slow queries")
            for query_info in slow_queries:
                logger.warning(
                    f"Slow query: {query_info['mean_exec_time']:.2f}ms avg, "
                    f"{query_info['calls']} calls"
                )

        return slow_queries

# Query optimization examples
class OptimizedQueries:
    """
    Optimized queries for common operations
    """
```

```python
    @staticmethod
    def get_company_with_ratios(ticker: str) -> str:
        """
        Single query to get company data with latest ratios
        """
        return """
        SELECT
            s.*,
            fr.pe_ratio, fr.pb_ratio, fr.roe, fr.debt_to_equity,
            fr.altman_z_score, fr.calculation_date as ratios_date
        FROM stocks s
        LEFT JOIN LATERAL (
            SELECT * FROM financial_ratios fr2
            WHERE fr2.ticker = s.ticker
            ORDER BY fr2.calculation_date DESC
            LIMIT 1
        ) fr ON true
        WHERE s.ticker = %s
        """

    @staticmethod
    def get_industry_companies_with_ratios(industry: str) -> str:
        """
        Optimized query for industry comparison
        """
        return """
        SELECT
            s.ticker, s.company_name, s.market_cap,
            fr.pe_ratio, fr.pb_ratio, fr.roe
        FROM stocks s
        JOIN mv_latest_company_metrics fr ON s.ticker = fr.ticker
        WHERE s.industry = %s
            AND s.market_cap > 100000000
            AND fr.pe_ratio IS NOT NULL
        ORDER BY s.market_cap DESC
        """

    @staticmethod
    def get_priority_companies_for_update() -> str:
        """
        Optimized priority selection query
        """
        return """
        WITH company_priorities AS (
```

```
        SELECT
            ticker,
            CASE
                WHEN next_earnings_date <= CURRENT_DATE + INTERVAL '7 days' THEN 5
                WHEN fundamentals_last_update IS NULL THEN 4
                WHEN fundamentals_last_update < CURRENT_DATE - INTERVAL '30 days' THEN 3
                WHEN fundamentals_last_update < CURRENT_DATE - INTERVAL '90 days' THEN 2
                ELSE 1
            END as priority,
            market_cap,
            fundamentals_last_update
        FROM stocks
        WHERE market_cap > 100000000
    )
    SELECT ticker
    FROM company_priorities
    WHERE priority >= %s
    ORDER BY priority DESC, market_cap DESC, fundamentals_last_update ASC NULLS FIRST
    LIMIT %s
    """
```

BACK-017: Monitoring & Alerting Dashboard

Type: Backend Task
Priority: Medium
Story Points: 13
Assignee: Backend Developer

Description: Build comprehensive monitoring for data pipeline health and API performance per professor's success metrics.

Acceptance Criteria:

- [ ] Track API success/failure rates (95% target)
- [ ] Monitor data freshness per company (<5% stale data target)
- [ ] Alert on calculation errors or data quality issues
- [ ] Weekly data quality reports
- [ ] Performance dashboard with professor's KPIs

PROFESSOR'S SUCCESS METRICS TO MONITOR:

- 95% daily success rate for priority companies

- <5% data staleness (companies with 30+ day old data)
- <1 hour total execution time for daily updates
- 99.9% uptime for database access

Implementation:

python

```python
class MonitoringDashboard:

    PROFESSOR_TARGETS = {
        'daily_success_rate': 0.95,      # 95% success rate
        'max_stale_data_pct': 0.05,      # <5% stale data
        'max_daily_execution_hours': 1,   # <1 hour daily updates
        'database_uptime_target': 0.999   # 99.9% uptime
    }

    def __init__(self):
        self.metrics_collector = MetricsCollector()
        self.alerting_service = AlertingService()

    def collect_daily_metrics(self) -> dict:
        """
        Collect all daily metrics per professor's requirements
        """
        metrics = {}

        # 1. API Success Rate
        api_metrics = self.metrics_collector.get_api_metrics_last_24h()
        metrics['api_success_rate'] = self.calculate_success_rate(api_metrics)

        # 2. Data Freshness
        freshness_metrics = self.metrics_collector.get_data_freshness_metrics()
        metrics['stale_data_percentage'] = self.calculate_stale_data_percentage(freshness_metri

        # 3. Execution Time
        execution_metrics = self.metrics_collector.get_execution_time_metrics()
        metrics['daily_execution_time_hours'] = execution_metrics['total_time_hours']

        # 4. Database Performance
        db_metrics = self.metrics_collector.get_database_metrics()
        metrics['database_uptime'] = db_metrics['uptime_percentage']
        metrics['avg_query_time_ms'] = db_metrics['avg_query_time']

        # 5. Data Quality Scores
        quality_metrics = self.metrics_collector.get_data_quality_metrics()
        metrics['avg_data_quality_score'] = quality_metrics['average_score']

        # Check against professor's targets
        metrics['targets_met'] = self.check_targets_compliance(metrics)
```

```
        return metrics

    def calculate_success_rate(self, api_metrics: dict) -> float:
        """
        Calculate overall API success rate
        """
        total_calls = sum(provider['total_calls'] for provider in api_metrics.values())
        successful_calls = sum(provider['successful_calls'] for provider in api_metrics.values(

        if total_calls == 0:
            return 1.0

        return successful_calls / total_calls

    def calculate_stale_data_percentage(self, freshness_metrics: dict) -> float:
        """
        Calculate percentage of companies with stale data (>30 days)
        """
        total_companies = freshness_metrics['total_companies']
        stale_companies = freshness_metrics['companies_over_30_days']

        if total_companies == 0:
            return 0.0

        return stale_companies / total_companies

    def check_targets_compliance(self, metrics: dict) -> dict:
        """
        Check if metrics meet professor's targets
        """
        compliance = {}

        compliance['api_success_rate'] = {
            'target': self.PROFESSOR_TARGETS['daily_success_rate'],
            'actual': metrics['api_success_rate'],
            'met': metrics['api_success_rate'] >= self.PROFESSOR_TARGETS['daily_success_rate']
        }

        compliance['stale_data'] = {
            'target': self.PROFESSOR_TARGETS['max_stale_data_pct'],
            'actual': metrics['stale_data_percentage'],
            'met': metrics['stale_data_percentage'] <= self.PROFESSOR_TARGETS['max_stale_data_p
        }
```

```
        compliance['execution_time'] = {
            'target': self.PROFESSOR_TARGETS['max_daily_execution_hours'],
            'actual': metrics['daily_execution_time_hours'],
            'met': metrics['daily_execution_time_hours'] <= self.PROFESSOR_TARGETS['max_daily_e
        }

        compliance['database_uptime'] = {
            'target': self.PROFESSOR_TARGETS['database_uptime_target'],
            'actual': metrics['database_uptime'],
            'met': metrics['database_uptime'] >= self.PROFESSOR_TARGETS['database_uptime_target
        }

        # Overall compliance
        compliance['overall'] = all(item['met'] for item in compliance.values())

        return compliance

    def generate_daily_report(self) -> dict:
        """
        Generate daily monitoring report
        """
        metrics = self.collect_daily_metrics()

        report = {
            'date': datetime.now().date(),
            'metrics': metrics,
            'summary': {
                'overall_health': 'Good' if metrics['targets_met']['overall'] else 'Issues Dete
                'critical_issues': [],
                'warnings': [],
                'recommendations': []
            }
        }

        # Identify issues and recommendations
        self.analyze_metrics_and_add_recommendations(metrics, report)

        # Send alerts if needed
        if not metrics['targets_met']['overall']:
            self.alerting_service.send_daily_summary_alert(report)

        return report

    def analyze_metrics_and_add_recommendations(self, metrics: dict, report: dict):
```

```
    """
    Analyze metrics and add recommendations
    """
    compliance = metrics['targets_met']

    # API Success Rate Issues
    if not compliance['api_success_rate']['met']:
        actual_rate = compliance['api_success_rate']['actual']
        report['summary']['critical_issues'].append(
            f"API success rate {actual_rate:.1%} below target {compliance['api_success_rate
        )
        report['summary']['recommendations'].append(
            "Investigate API failures and consider adjusting rate limits or fallback strate
        )

    # Data Staleness Issues
    if not compliance['stale_data']['met']:
        stale_pct = compliance['stale_data']['actual']
        report['summary']['warnings'].append(
            f"Stale data percentage {stale_pct:.1%} above target {compliance['stale_data'][
        )
        report['summary']['recommendations'].append(
            "Increase update frequency for companies with old data"
        )

    # Execution Time Issues
    if not compliance['execution_time']['met']:
        actual_time = compliance['execution_time']['actual']
        report['summary']['warnings'].append(
            f"Daily execution time {actual_time:.1f}h exceeds {compliance['execution_time']
        )
        report['summary']['recommendations'].append(
            "Optimize database queries or reduce batch sizes"
        )

    # Database Performance Issues
    if metrics['avg_query_time_ms'] > 100:  # Professor's 100ms requirement
        report['summary']['warnings'].append(
            f"Average query time {metrics['avg_query_time_ms']:.1f}ms exceeds 100ms target"
        )
        report['summary']['recommendations'].append(
            "Review slow queries and add missing indexes"
        )
```

```python
class AlertingService:
    """
    Email and Slack alerting for critical issues
    """

    def send_daily_summary_alert(self, report: dict):
        """
        Send daily summary alert if targets not met
        """
        subject = f"Data Pipeline Health Alert - {report['date']}"

        body = f"""
        Daily monitoring report for {report['date']}:

        Overall Health: {report['summary']['overall_health']}

        Critical Issues:
        {self.format_list(report['summary']['critical_issues'])}

        Warnings:
        {self.format_list(report['summary']['warnings'])}

        Recommendations:
        {self.format_list(report['summary']['recommendations'])}

        Detailed Metrics:
        - API Success Rate: {report['metrics']['api_success_rate']:.1%}
        - Stale Data: {report['metrics']['stale_data_percentage']:.1%}
        - Execution Time: {report['metrics']['daily_execution_time_hours']:.1f}h
        - Database Uptime: {report['metrics']['database_uptime']:.2%}

        Please review and take appropriate action.
        """

        self.email_service.send_alert(subject, body, priority='medium')

    def send_critical_alert(self, alert_type: str, details: dict):
        """
        Send immediate alert for critical issues
        """
        subject = f"CRITICAL: {alert_type}"

        body = f"""
        Critical issue detected in data pipeline:
```

```
        Type: {alert_type}
        Time: {datetime.now()}
        Details: {details}

        Immediate action required.
        """

        self.email_service.send_alert(subject, body, priority='high')
        # Also send to Slack if configured
        self.slack_service.send_alert(subject, body)

    def format_list(self, items: list) -> str:
        """Format list items for email"""
        if not items:
            return "None"
        return "\n".join(f"- {item}" for item in items)
```

📋 FINAL IMPLEMENTATION SUMMARY

SUCCESS METRICS (Professor's Requirements):

- 95% daily success rate for priority companies
- <5% data staleness (companies with 30+ day old data)
- <1 hour total execution time for daily updates
- 99.9% uptime for database access
- <100ms average query response time

RATE LIMIT MANAGEMENT:

- **Monday/Wednesday**: 400 companies via Yahoo Finance (2000/hour limit)
- **Tuesday/Thursday**: 300 companies via mixed providers
- **Friday**: 100 companies via Alpha Vantage (5/min limit)
- **Total**: ~1300 companies updated weekly with priority system

DATA QUALITY ASSURANCE:

- Multi-provider failover cascade (Yahoo → Finnhub → Alpha Vantage)
- Comprehensive validation rules with automatic corrections
- Confidence scoring (80+ High, 60-79 Medium, <60 Low)

- Score adjustments: -15% for low confidence, -5% for medium

PERFORMANCE TARGETS:

- All database queries <100ms
- Industry benchmarks cached for 24 hours
- Peer data cached for 7 days
- Materialized views refreshed daily
- Composite indexes on all query patterns

This comprehensive implementation provides all the detailed specifications needed for the development team to build a robust, scalable value investing analysis system that meets the professor's exact requirements while maintaining high performance and reliability standards.

📋 TECHNICAL REQUIREMENTS

Environment Setup:

- Python 3.11+
- PostgreSQL 15+ (Railway)
- Redis for caching
- Celery for task queues
- APScheduler for cron jobs

External APIs:

- Yahoo Finance (primary)
- Finnhub (secondary)
- Alpha Vantage (tertiary)

Rate Limit Strategy:

- Distribute 1500 companies across 5 weekdays
- Monday/Wednesday: 400 companies each (Yahoo)
- Tuesday/Thursday: 300 companies each (mixed)
- Friday: 100 companies (Alpha Vantage)

Data Update Frequency:

- **Real-time**: None (frontend reads from DB)
- **Daily**: Fundamental data for priority companies
- **Weekly**: Industry benchmarks recalculation
- **Monthly**: Full data quality audit

Error Recovery:

- 3 retry attempts with exponential backoff
- Dead letter queue for persistent failures
- Email alerts for critical system errors
- Graceful degradation when APIs unavailable

📋 DEFINITION OF DONE

For each task to be considered complete:

1. **Code Quality:**
   - ☐ Unit tests with >80% coverage
   - ☐ Integration tests for API calls
   - ☐ Code review completed
   - ☐ Documentation updated
2. **Performance:**
   - ☐ Database queries execute in <100ms
   - ☐ API calls respect rate limits
   - ☐ Memory usage stable over 24hr period
3. **Reliability:**
   - ☐ Error handling for all failure modes
   - ☐ Logging for debugging and monitoring
   - ☐ Graceful degradation when dependencies fail
4. **Data Quality:**
   - ☐ Validation rules implemented
   - ☐ Data consistency checks pass
   - ☐ Manual spot-checking of calculated ratios

Success Metrics:

- 95% daily success rate for priority companies

- <5% data staleness (companies with 30+ day old data)
- <1 hour total execution time for daily updates
- 99.9% uptime for database access