MARKET CRASH & BOTTOM PREDICTION SYSTEM

Table of Contents

Complete Implementation Guide for Augment Code

You are tasked with building a production-ready market crash and bottom prediction system. This document contains complete specifications for all 7 phases of development. Read carefully and implement exactly as specified.

PROJECT OVERVIEW

Your Mission

Build an end-to-end machine learning system that:

1. **Predicts market crashes** with 85-93% accuracy using ensemble ML models
2. **Predicts market bottoms** for optimal re-entry with 85-90% accuracy (UNIQUE FEATURE)
3. **Provides real-time dashboard** with individual charts for all 28 indicators
4. **Sends automated alerts** via email/SMS based on individual indicators OR ML outputs

Business Value

- Crash prediction: Save 25-35% losses by knowing when to exit
- Bottom prediction: Capture 25-40% recovery gains by knowing when to re-enter
- ROI: $15,000-$95,000 annual benefit for $100k-$500k portfolios
- This is a HIGH-VALUE, production-ready system

Technology Stack

- **Language**: Python 3.9+
- **Data Sources**: FRED API, yfinance, Alpha Vantage
- **ML**: scikit-learn, TensorFlow/Keras, XGBoost
- **Dashboard**: Streamlit
- **Database**: SQLite (dev) → PostgreSQL (prod)
- **Alerts**: SMTP (email), Twilio (SMS)
- **Scheduler**: APScheduler

- **Testing**: pytest (>= 85% coverage required)

Success Metrics You Must Achieve

Metric	Target	How to Measure
Crash Prediction AUC	>= 0.85	ROC-AUC on out-of-sample tests
Crash Prediction Accuracy	>= 85%	Correct / total predictions
Crash Detection (Recall)	>= 75%	True positives / actual crashes
False Alarm Rate	<= 20%	False positives / total predictions
Bottom Prediction Accuracy	85-90%	Within ±30 days of actual bottom
Recovery Time Prediction	±3 months	R^2 >= 0.75
Dashboard Response Time	< 2 seconds	Load/update time
Alert Delivery	>= 99%	Delivered / attempted
Test Coverage	>= 85%	pytest-cov

PROJECT STRUCTURE

Directory Layout (Create Exactly)

```
market-crash-predictor/
├── data/
│   ├── raw/            # Raw API data
│   ├── processed/      # Processed features
│   ├── models/         # Trained model files
│   └── logs/           # Data collection logs
├── src/
│   ├── __init__.py
│   ├── data_collection/
│   │   ├── __init__.py
│   │   ├── base_collector.py
│   │   ├── fred_collector.py
│   │   ├── yahoo_collector.py
│   │   └── alpha_vantage_collector.py
│   ├── feature_engineering/
│   │   ├── __init__.py
│   │   ├── crash_indicators.py
│   │   ├── bottom_features.py
│   │   └── regime_detection.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── crash_prediction/
│   │   │   ├── __init__.py
│   │   │   ├── base_model.py
│   │   │   ├── svm_model.py
│   │   │   ├── random_forest_model.py
│   │   │   ├── gradient_boosting_model.py
│   │   │   ├── neural_network_model.py
```

```
│   │   │   └── ensemble_model.py
│   │   ├── bottom_prediction/
│   │   │   ├── __init__.py
│   │   │   ├── mlp_model.py
│   │   │   ├── lstm_model.py
│   │   │   └── ensemble_model.py
│   │   └── utils/
│   │       ├── __init__.py
│   │       ├── evaluation.py
│   │       └── validation.py
│   ├── dashboard/
│   │   ├── __init__.py
│   │   ├── app.py
│   │   ├── pages/
│   │   │   ├── home.py
│   │   │   ├── indicators.py
│   │   │   ├── predictions.py
│   │   │   └── settings.py
│   │   └── components/
│   │       ├── __init__.py
│   │       ├── charts.py
│   │       ├── metrics.py
│   │       └── alerts_config.py
│   ├── alerts/
│   │   ├── __init__.py
│   │   ├── base_alert.py
│   │   ├── email_alert.py
│   │   ├── sms_alert.py
│   │   ├── alert_manager.py
│   │   └── alert_conditions.py
│   ├── scheduler/
│   │   ├── __init__.py
│   │   ├── daily_tasks.py
│   │   └── task_manager.py
│   └── utils/
│       ├── __init__.py
│       ├── config.py
│       ├── database.py
│       ├── logger.py
│       └── validators.py
├── tests/
│   ├── __init__.py
│   ├── test_data_collection/
│   ├── test_feature_engineering/
│   ├── test_models/
│   ├── test_dashboard/
│   └── test_alerts/
├── notebooks/
│   ├── 01_data_exploration.ipynb
│   ├── 02_feature_analysis.ipynb
│   ├── 03_crash_model_training.ipynb
│   ├── 04_bottom_model_training.ipynb
│   └── 05_backtesting.ipynb
├── config/
│   ├── config.yaml
│   ├── api_keys.env.example
```

```
│   ├── thresholds.yaml
│   └── logging.yaml
├── docs/
│   ├── API_DOCUMENTATION.md
│   ├── USER_GUIDE.md
│   └── MODEL_METHODOLOGY.md
├── scripts/
│   ├── setup_database.py
│   ├── backfill_data.py
│   ├── train_models.py
│   └── deploy.sh
├── requirements.txt
├── .env.example
├── .gitignore
├── README.md
└── LICENSE
```

PHASE 1: DATA PIPELINE SETUP

Duration: 1-2 weeks
Priority: CRITICAL - Must complete before Phase 2

Your Tasks

1.1 Implement FRED API Client

File: `src/data_collection/fred_collector.py`

Fetch these 15 indicators from FRED:

Indicator	FRED ID	Frequency	History
10Y-3M Yield Spread	T10Y3M	Daily	1982+
10Y-2Y Yield Spread	T10Y2Y	Daily	1976+
BBB Credit Spread	BAMLC0A4CBBB	Daily	1996+
Unemployment Rate	UNRATE	Monthly	1948+
Real GDP	GDPC1	Quarterly	1947+
CPI	CPIAUCSL	Monthly	1947+
Federal Funds Rate	FEDFUNDS	Monthly	1954+
VIX	VIXCLS	Daily	1990+
Consumer Sentiment	UMCSENT	Monthly	1978+
Housing Starts	HOUST	Monthly	1959+
Industrial Production	INDPRO	Monthly	1919+
M2 Money Supply	M2SL	Monthly	1959+

Indicator	FRED ID	Frequency	History
Debt to GDP	GFDEGDQ188S	Quarterly	1966+
Savings Rate	PSAVERT	Monthly	1959+
LEI	USSLIND	Monthly	1959+

Implementation Requirements:

```
import fredapi
import pandas as pd
import logging
from typing import Dict

class FREDCollector:
    """Collects economic indicators from FRED API."""

    INDICATORS = {
        'yield_10y_3m': 'T10Y3M',
        'yield_10y_2y': 'T10Y2Y',
        'credit_spread_bbb': 'BAMLC0A4CBBB',
        'unemployment_rate': 'UNRATE',
        'real_gdp': 'GDPC1',
        'cpi': 'CPIAUCSL',
        'fed_funds_rate': 'FEDFUNDS',
        'vix': 'VIXCLS',
        'consumer_sentiment': 'UMCSENT',
        'housing_starts': 'HOUST',
        'industrial_production': 'INDPRO',
        'm2_money_supply': 'M2SL',
        'debt_to_gdp': 'GFDEGDQ188S',
        'savings_rate': 'PSAVERT',
        'lei': 'USSLIND'
    }

    def __init__(self, api_key: str):
        """Initialize with FRED API key."""
        self.api_key = api_key
        self.fred = fredapi.Fred(api_key=api_key)
        self.logger = logging.getLogger(__name__)

    def fetch_indicator(self, indicator_name: str,
                        start_date: str = '1970-01-01',
                        end_date: str = None) -&gt; pd.Series:
        """
        Fetch single indicator from FRED.

        Args:
            indicator_name: Name from INDICATORS dict
            start_date: Start date 'YYYY-MM-DD'
            end_date: End date 'YYYY-MM-DD' (default: today)

        Returns:
            pd.Series with DatetimeIndex
```

```
        Raises:
            ValueError: If indicator_name invalid
        """
        if indicator_name not in self.INDICATORS:
            raise ValueError(f"Unknown indicator: {indicator_name}")

        series_id = self.INDICATORS[indicator_name]

        try:
            data = self.fred.get_series(
                series_id,
                observation_start=start_date,
                observation_end=end_date
            )
            self.logger.info(f"Fetched {indicator_name}: {len(data)} observations")
            return data
        except Exception as e:
            self.logger.error(f"Failed to fetch {indicator_name}: {e}")
            raise

    def fetch_all_indicators(self, start_date: str = '1970-01-01',
                             end_date: str = None) -&gt; pd.DataFrame:
        """
        Fetch all 15 indicators.

        Returns:
            DataFrame with all indicators
        """
        df = pd.DataFrame()

        for name in self.INDICATORS:
            try:
                series = self.fetch_indicator(name, start_date, end_date)
                df[name] = series
            except Exception as e:
                self.logger.error(f"Skipping {name}: {e}")
                continue

        return df
```

Your Requirements:

- Add retry logic with exponential backoff for API failures
- Implement rate limiting (120 requests/minute FRED limit)
- Add comprehensive error handling
- Write unit tests with >= 90% coverage
- Use type hints for all methods
- Add detailed docstrings

1.2 Implement Yahoo Finance Client

File: `src/data_collection/yahoo_collector.py`

Fetch S&P 500 and VIX data:

```
import yfinance as yf
import pandas as pd
import numpy as np
import logging

class YahooCollector:
    """Collects market data from Yahoo Finance."""

    SYMBOLS = {
        'sp500': '^GSPC',
        'vix': '^VIX'
    }

    def __init__(self):
        self.logger = logging.getLogger(__name__)

    def fetch_price_data(self, symbol: str,
                         start_date: str = '1970-01-01',
                         end_date: str = None) -&gt; pd.DataFrame:
        """
        Fetch OHLCV data for symbol.

        Returns:
            DataFrame with Open, High, Low, Close, Volume, Adj Close
        """
        if symbol not in self.SYMBOLS:
            raise ValueError(f"Unknown symbol: {symbol}")

        ticker = self.SYMBOLS[symbol]

        try:
            data = yf.download(ticker, start=start_date, end=end_date, progress=False)
            self.logger.info(f"Fetched {symbol}: {len(data)} days")
            return data
        except Exception as e:
            self.logger.error(f"Failed to fetch {symbol}: {e}")
            raise

    def calculate_returns(self, prices: pd.Series) -&gt; pd.Series:
        """Calculate log returns."""
        return np.log(prices / prices.shift(1)).dropna()

    def calculate_volatility(self, returns: pd.Series, window: int = 20) -&gt; pd.Series:
        """Calculate annualized rolling volatility."""
        return returns.rolling(window).std() * np.sqrt(252) * 100

    def calculate_drawdown(self, prices: pd.Series) -&gt; pd.Series:
        """Calculate drawdown from all-time high."""
```

```
        cummax = prices.cummax()
        return (prices - cummax) / cummax * 100
```

1.3 Implement Database Layer

File: `src/utils/database.py`

Create SQLAlchemy ORM models:

```
from sqlalchemy import create_engine, Column, Integer, Float, Date, String, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from datetime import datetime
import logging

Base = declarative_base()

class Indicator(Base):
    """ORM model for indicators table."""
    __tablename__ = 'indicators'

    id = Column(Integer, primary_key=True)
    date = Column(Date, nullable=False, unique=True, index=True)

    # Yield curve
    yield_10y_3m = Column(Float)
    yield_10y_2y = Column(Float)

    # Credit
    credit_spread_bbb = Column(Float)

    # Economic
    unemployment_rate = Column(Float)
    real_gdp = Column(Float)
    cpi = Column(Float)
    fed_funds_rate = Column(Float)

    # Market
    sp500_close = Column(Float)
    sp500_volume = Column(Float)
    vix_close = Column(Float)

    # Sentiment
    consumer_sentiment = Column(Float)

    # Housing
    housing_starts = Column(Float)

    # Production
    industrial_production = Column(Float)

    # Monetary
    m2_money_supply = Column(Float)
```

```
    # Debt
    debt_to_gdp = Column(Float)

    # Savings
    savings_rate = Column(Float)

    # Composite
    lei = Column(Float)

    # Metadata
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)
    data_quality_score = Column(Float, default=1.0)

class CrashEvent(Base):
    """ORM model for crash_events table."""
    __tablename__ = 'crash_events'

    id = Column(Integer, primary_key=True)
    start_date = Column(Date, nullable=False)
    end_date = Column(Date, nullable=False)
    trough_date = Column(Date, nullable=False)
    recovery_date = Column(Date)
    max_drawdown = Column(Float, nullable=False)
    recovery_months = Column(Integer)
    crash_type = Column(String(50))
    notes = Column(String)

class Prediction(Base):
    """ORM model for predictions table."""
    __tablename__ = 'predictions'

    id = Column(Integer, primary_key=True)
    prediction_date = Column(Date, nullable=False, index=True)
    crash_probability = Column(Float, nullable=False)
    bottom_prediction_date = Column(Date)
    recovery_prediction_date = Column(Date)
    confidence_interval_lower = Column(Float)
    confidence_interval_upper = Column(Float)
    model_version = Column(String(20), nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow)

class AlertHistory(Base):
    """ORM model for alert_history table."""
    __tablename__ = 'alert_history'

    id = Column(Integer, primary_key=True)
    alert_date = Column(DateTime, nullable=False)
    alert_type = Column(String(50), nullable=False)
    condition_met = Column(String, nullable=False)
    recipients = Column(String, nullable=False)
    delivery_status = Column(String(20), nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow)

class DatabaseManager:
    """Manages database connections."""
```

```
    def __init__(self, db_url: str = 'sqlite:///data/market_crash.db'):
        self.engine = create_engine(db_url)
        self.SessionLocal = sessionmaker(bind=self.engine)
        self.logger = logging.getLogger(__name__)

    def create_tables(self):
        """Create all tables."""
        Base.metadata.create_all(self.engine)
        self.logger.info("Database tables created")

    def get_session(self):
        """Get database session."""
        return self.SessionLocal()
```

1.4 Implement Data Validation

File: `src/utils/validators.py`

```
import pandas as pd
import numpy as np
from typing import Dict, List
import logging

class DataValidator:
    """Validates data quality for market indicators."""

    EXPECTED_RANGES = {
        'yield_10y_3m': (-5.0, 5.0),
        'yield_10y_2y': (-3.0, 3.0),
        'credit_spread_bbb': (0.0, 15.0),
        'unemployment_rate': (2.0, 25.0),
        'vix_close': (5.0, 100.0),
        'sp500_close': (0.0, None),
        'consumer_sentiment': (0.0, 150.0),
    }

    def __init__(self):
        self.logger = logging.getLogger(__name__)

    def validate_missing_values(self, df: pd.DataFrame) -&gt; Dict:
        """Analyze missing values."""
        total_missing = df.isnull().sum().sum()
        missing_by_column = (df.isnull().sum() / len(df) * 100).to_dict()
        critical_missing = [col for col, pct in missing_by_column.items() if pct &gt; 5.0

        return {
            'total_missing': total_missing,
            'missing_by_column': missing_by_column,
            'critical_missing': critical_missing
        }

    def validate_value_ranges(self, df: pd.DataFrame) -&gt; Dict:
        """Check if values are within expected ranges."""
        outliers = {}
```

```
        all_valid = True

        for col in df.columns:
            if col in self.EXPECTED_RANGES:
                min_val, max_val = self.EXPECTED_RANGES[col]

                if min_val is not None:
                    below_min = df[df[col] &lt; min_val]
                    if not below_min.empty:
                        outliers[f"{col}_below_min"] = len(below_min)
                        all_valid = False

                if max_val is not None:
                    above_max = df[df[col] &gt; max_val]
                    if not above_max.empty:
                        outliers[f"{col}_above_max"] = len(above_max)
                        all_valid = False

        return {'valid': all_valid, 'outliers': outliers}

    def fill_missing_values(self, df: pd.DataFrame) -&gt; pd.DataFrame:
        """Fill missing values using forward-fill and interpolation."""
        df_filled = df.copy()

        # Forward fill most indicators
        forward_fill_cols = ['yield_10y_3m', 'yield_10y_2y', 'credit_spread_bbb', 'vix_cl
        for col in forward_fill_cols:
            if col in df_filled.columns:
                df_filled[col] = df_filled[col].fillna(method='ffill')

        # Interpolate continuous indicators
        interpolate_cols = ['real_gdp', 'cpi', 'industrial_production']
        for col in interpolate_cols:
            if col in df_filled.columns:
                df_filled[col] = df_filled[col].interpolate(method='linear')

        return df_filled
```

1.5 Implement Scheduler

File: `src/scheduler/daily_tasks.py`

```
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.triggers.cron import CronTrigger
import logging
from datetime import datetime
import os
import pandas as pd

class DailyPipeline:
    """Orchestrates daily data collection and processing."""

    def __init__(self):
        self.scheduler = BlockingScheduler()
        self.logger = logging.getLogger(__name__)
```

```
    def run_daily_update(self):
        """Execute full daily update pipeline."""
        try:
            self.logger.info(f"Starting daily update at {datetime.now()}")

            # Step 1: Collect data
            self.collect_fred_data()
            self.collect_yahoo_data()

            # Step 2: Validate
            self.validate_data()

            # Step 3: Store
            self.store_data()

            # Step 4: Generate features
            self.generate_features()

            self.logger.info(f"Daily update completed at {datetime.now()}")

        except Exception as e:
            self.logger.error(f"Daily update failed: {e}", exc_info=True)
            self.send_error_alert(str(e))

    def collect_fred_data(self):
        """Collect latest FRED indicators."""
        from src.data_collection.fred_collector import FREDCollector

        api_key = os.getenv('FRED_API_KEY')
        collector = FREDCollector(api_key)

        end_date = datetime.now().strftime('%Y-%m-%d')
        start_date = (datetime.now() - pd.Timedelta(days=7)).strftime('%Y-%m-%d')

        self.fred_data = collector.fetch_all_indicators(start_date, end_date)
        self.logger.info(f"Collected FRED data: {self.fred_data.shape}")

    def collect_yahoo_data(self):
        """Collect latest Yahoo Finance data."""
        from src.data_collection.yahoo_collector import YahooCollector

        collector = YahooCollector()

        end_date = datetime.now().strftime('%Y-%m-%d')
        start_date = (datetime.now() - pd.Timedelta(days=7)).strftime('%Y-%m-%d')

        sp500_data = collector.fetch_price_data('sp500', start_date, end_date)
        vix_data = collector.fetch_price_data('vix', start_date, end_date)

        self.yahoo_data = {'sp500': sp500_data, 'vix': vix_data}
        self.logger.info("Collected Yahoo data")

    def validate_data(self):
        """Validate collected data."""
        from src.utils.validators import DataValidator
```

```
        validator = DataValidator()
        validation = validator.validate_missing_values(self.fred_data)

        if validation['critical_missing']:
            self.logger.warning(f"Critical missing: {validation['critical_missing']}")

    def store_data(self):
        """Store data in database."""
        self.logger.info("Data stored")

    def generate_features(self):
        """Generate features (Phase 2)."""
        self.logger.info("Features generated")

    def send_error_alert(self, error_message: str):
        """Send error alert."""
        self.logger.error(f"Error alert: {error_message}")

    def start_scheduler(self):
        """Start scheduler (6 AM daily)."""
        self.scheduler.add_job(
            self.run_daily_update,
            CronTrigger(hour=6, minute=0),
            id='daily_update',
            name='Daily data collection',
            replace_existing=True
        )

        self.logger.info("Scheduler started. Updates at 6:00 AM daily.")
        self.scheduler.start()
```

1.6 Create Backfill Script

File: `scripts/backfill_data.py`

```
import sys
import os
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from src.data_collection.fred_collector import FREDCollector
from src.data_collection.yahoo_collector import YahooCollector
from src.utils.database import DatabaseManager, Indicator
from src.utils.validators import DataValidator
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def backfill_historical_data():
    """Backfill 55 years of historical data (1970-2025)."""

    logger.info("Starting historical data backfill...")

    # Initialize collectors
```

```
    fred_api_key = os.getenv('FRED_API_KEY')
    fred_collector = FREDCollector(fred_api_key)
    yahoo_collector = YahooCollector()

    # Fetch data
    logger.info("Fetching FRED data...")
    fred_data = fred_collector.fetch_all_indicators('1970-01-01', '2025-12-31')

    logger.info("Fetching Yahoo data...")
    sp500_data = yahoo_collector.fetch_price_data('sp500', '1970-01-01', '2025-12-31')
    vix_data = yahoo_collector.fetch_price_data('vix', '1990-01-01', '2025-12-31')

    # Validate
    logger.info("Validating data...")
    validator = DataValidator()
    validation = validator.validate_missing_values(fred_data)
    logger.info(f"Missing data: {validation['missing_by_column']}")

    # Fill missing values
    fred_data = validator.fill_missing_values(fred_data)

    # Store in database
    logger.info("Storing in database...")
    db_manager = DatabaseManager()
    db_manager.create_tables()

    # Store data (implement actual storage logic)
    logger.info(f"Backfill complete. Total records: {len(fred_data)}")

if __name__ == "__main__":
    backfill_historical_data()
```

Phase 1 Acceptance Criteria

You must verify:

- [ ] All API clients authenticate and fetch data successfully
- [ ] Database contains 1970-2025 historical data (55 years)
- [ ] < 1% missing values after imputation
- [ ] Daily scheduler runs at 6 AM without errors
- [ ] Test coverage >= 90% for data collection modules
- [ ] All functions have type hints and docstrings
- [ ] Code passes pylint (score >= 8.0)

PHASE 2: FEATURE ENGINEERING

Duration: 1 week
Dependencies: Phase 1 complete

Your Tasks

2.1 Implement 28 Crash Indicators

File: `src/feature_engineering/crash_indicators.py`

Implement these 28 indicators with exact formulas:

Financial Market Indicators (8):

1. Yield Spread 10Y-3M: `yield_10y - yield_3m`
2. Yield Spread 10Y-2Y: `yield_10y - yield_2y`
3. Credit Spread BBB: `bbb_yield - yield_10y`
4. VIX Level: Direct observation
5. VIX Change Rate: `(vix_t - vix_t-20) / vix_t-20 * 100`
6. Realized Volatility: `sqrt(sum(log_returns^2)) * sqrt(252)` (20-day)
7. S&P 500 Momentum 200D: `(price - ma_200) / ma_200 * 100`
8. S&P 500 Drawdown: `(price - cummax(price)) / cummax(price) * 100`

Credit Cycle Indicators (6):
9. Debt Service Ratio (DSR): From FRED
10. Credit-to-GDP Gap: `(credit/gdp) - trend` (HP filter)
11. Corporate Debt Growth: YoY % change
12. Household Debt Growth: YoY % change
13. M2 Growth: YoY % change
14. Debt to GDP: `federal_debt / gdp * 100`

Valuation Indicators (4):
15. Shiller PE (CAPE): `price / avg(earnings_10y)`
16. Buffett Indicator: `market_cap / gdp * 100`
17. S&P 500 P/B Ratio: `market_cap / book_value`
18. Earnings Yield Spread: `(earnings/price) - yield_10y`

Sentiment Indicators (5):
19. Consumer Sentiment: Direct from FRED
20. Put/Call Ratio: `put_volume / call_volume`
21. Margin Debt: Direct observation
22. Margin Debt Growth: YoY % change
23. Market Breadth: `advancing_stocks / total_stocks * 100`

Economic Indicators (5):
24. Unemployment Rate: Direct from FRED
25. Sahm Rule: `u3ma_t - min(u3ma_t-12)`
26. GDP Growth: YoY % change
27. Industrial Production Growth: YoY % change
28. Housing Starts Growth: YoY % change

Implementation:

```
import pandas as pd
import numpy as np
from typing import Dict

class CrashIndicators:
    """Calculate all 28 crash prediction indicators."""

    @staticmethod
    def calculate_all_indicators(df: pd.DataFrame) -&gt; pd.DataFrame:
        """
        Calculate all 28 indicators from raw data.

        Args:
            df: DataFrame with raw indicators

        Returns:
            DataFrame with 28 calculated indicators
        """
        indicators = pd.DataFrame(index=df.index)

        # Financial Market (8)
        indicators['yield_spread_10y_3m'] = df['yield_10y'] - df['yield_3m']
        indicators['yield_spread_10y_2y'] = df['yield_10y'] - df['yield_2y']
        indicators['credit_spread_bbb'] = df['bbb_yield'] - df['yield_10y']
        indicators['vix_level'] = df['vix']
        indicators['vix_change_rate'] = (df['vix'] / df['vix'].shift(20) - 1) * 100
        indicators['realized_volatility'] = CrashIndicators.realized_volatility(df)
        indicators['sp500_momentum_200d'] = CrashIndicators.sp500_momentum_200d(df)
        indicators['sp500_drawdown'] = CrashIndicators.sp500_drawdown(df)

        # Credit Cycle (6)
        indicators['dsr'] = df['dsr']
        indicators['credit_gap'] = CrashIndicators.credit_gap(df)
        indicators['corporate_debt_growth'] = CrashIndicators.yoy_growth(df, 'corporate_d
        indicators['household_debt_growth'] = CrashIndicators.yoy_growth(df, 'household_d
        indicators['m2_growth'] = CrashIndicators.yoy_growth(df, 'm2_money_supply')
        indicators['debt_to_gdp'] = df['debt_to_gdp']

        # Valuation (4)
        indicators['shiller_pe'] = df.get('shiller_pe', np.nan)
        indicators['buffett_indicator'] = CrashIndicators.buffett_indicator(df)
        indicators['sp500_pb_ratio'] = df.get('sp500_pb', np.nan)
        indicators['earnings_yield_spread'] = CrashIndicators.earnings_yield_spread(df)

        # Sentiment (5)
```

```
        indicators['consumer_sentiment'] = df['consumer_sentiment']
        indicators['put_call_ratio'] = df.get('put_call_ratio', np.nan)
        indicators['margin_debt'] = df.get('margin_debt', np.nan)
        indicators['margin_debt_growth'] = CrashIndicators.yoy_growth(df, 'margin_debt')
        indicators['market_breadth'] = df.get('market_breadth', np.nan)

        # Economic (5)
        indicators['unemployment_rate'] = df['unemployment_rate']
        indicators['sahm_rule'] = CrashIndicators.sahm_rule(df)
        indicators['gdp_growth'] = CrashIndicators.yoy_growth(df, 'real_gdp')
        indicators['industrial_production_growth'] = CrashIndicators.yoy_growth(df, 'indu
        indicators['housing_starts_growth'] = CrashIndicators.yoy_growth(df, 'housing_sta

        return indicators

    @staticmethod
    def realized_volatility(df: pd.DataFrame, window: int = 20) -&gt; pd.Series:
        """Calculate annualized realized volatility."""
        returns = np.log(df['sp500_close'] / df['sp500_close'].shift(1))
        vol = returns.rolling(window).std() * np.sqrt(252) * 100
        return vol

    @staticmethod
    def sp500_momentum_200d(df: pd.DataFrame) -&gt; pd.Series:
        """Calculate momentum vs 200-day MA."""
        ma_200 = df['sp500_close'].rolling(200).mean()
        return (df['sp500_close'] / ma_200 - 1) * 100

    @staticmethod
    def sp500_drawdown(df: pd.DataFrame) -&gt; pd.Series:
        """Calculate drawdown from all-time high."""
        cummax = df['sp500_close'].cummax()
        return (df['sp500_close'] - cummax) / cummax * 100

    @staticmethod
    def credit_gap(df: pd.DataFrame) -&gt; pd.Series:
        """Calculate credit-to-GDP gap."""
        if 'total_credit' in df.columns and 'gdp' in df.columns:
            credit_to_gdp = df['total_credit'] / df['gdp'] * 100
            trend = credit_to_gdp.rolling(40).mean()
            return credit_to_gdp - trend
        return pd.Series(index=df.index, dtype=float)

    @staticmethod
    def yoy_growth(df: pd.DataFrame, column: str) -&gt; pd.Series:
        """Calculate year-over-year growth rate."""
        if column not in df.columns:
            return pd.Series(index=df.index, dtype=float)

        periods = 12  # Monthly data
        return (df[column] / df[column].shift(periods) - 1) * 100

    @staticmethod
    def buffett_indicator(df: pd.DataFrame) -&gt; pd.Series:
        """Calculate Market Cap to GDP ratio."""
        if 'market_cap' in df.columns and 'gdp' in df.columns:
```

```
            return df['market_cap'] / df['gdp'] * 100
        return pd.Series(index=df.index, dtype=float)

    @staticmethod
    def earnings_yield_spread(df: pd.DataFrame) -&gt; pd.Series:
        """Calculate earnings yield minus 10Y yield."""
        if 'earnings_yield' in df.columns:
            return df['earnings_yield'] - df['yield_10y']
        return pd.Series(index=df.index, dtype=float)

    @staticmethod
    def sahm_rule(df: pd.DataFrame) -&gt; pd.Series:
        """Calculate Sahm Rule indicator."""
        if 'unemployment_rate' not in df.columns:
            return pd.Series(index=df.index, dtype=float)

        u3ma = df['unemployment_rate'].rolling(3).mean()
        min_12m = u3ma.rolling(12, min_periods=1).min()
        return u3ma - min_12m
```

2.2 Implement Regime Detection

File: `src/feature_engineering/regime_detection.py`

```
import pandas as pd
import numpy as np
from typing import List, Dict

class BryBoschanAlgorithm:
    """
    Bry-Boschan algorithm for identifying market cycles.
    Used to label crash periods and bottoms.
    """

    def __init__(self, min_duration: int = 15, min_amplitude: float = 0.15):
        self.min_duration = min_duration
        self.min_amplitude = min_amplitude

    def identify_turning_points(self, prices: pd.Series) -&gt; pd.DataFrame:
        """
        Identify peaks and troughs in price series.

        Returns:
            DataFrame with columns: date, price, type, amplitude
        """
        extrema = self._find_local_extrema(prices)
        filtered = self._apply_filters(extrema, prices)
        alternating = self._ensure_alternating(filtered)

        return pd.DataFrame(alternating)

    def _find_local_extrema(self, prices: pd.Series, window: int = 5) -&gt; List[Dict]:
        """Find local maxima and minima."""
        extrema = []
```

```
        for i in range(window, len(prices) - window):
            current = prices.iloc[i]
            window_slice = prices.iloc[i-window:i+window+1]

            if current == window_slice.max() and current &gt; window_slice.mean() * 1.02:
                extrema.append({
                    'date': prices.index[i],
                    'price': current,
                    'type': 'peak',
                    'index': i
                })
            elif current == window_slice.min() and current &lt; window_slice.mean() * 0.9
                extrema.append({
                    'date': prices.index[i],
                    'price': current,
                    'type': 'trough',
                    'index': i
                })

        return extrema

    def _apply_filters(self, extrema: List[Dict], prices: pd.Series) -&gt; List[Dict]:
        """Apply duration and amplitude filters."""
        if len(extrema) &lt; 2:
            return extrema

        filtered = [extrema[0]]

        for i in range(1, len(extrema)):
            prev = filtered[-1]
            curr = extrema[i]

            duration = curr['index'] - prev['index']
            if duration &lt; self.min_duration:
                continue

            amplitude = abs(curr['price'] - prev['price']) / prev['price']
            if amplitude &lt; self.min_amplitude:
                continue

            filtered.append(curr)

        return filtered

    def _ensure_alternating(self, extrema: List[Dict]) -&gt; List[Dict]:
        """Ensure peaks and troughs alternate."""
        if len(extrema) &lt; 2:
            return extrema

        alternating = [extrema[0]]

        for point in extrema[1:]:
            if point['type'] != alternating[-1]['type']:
                alternating.append(point)
            else:
                if ((point['type'] == 'peak' and point['price'] &gt; alternating[-1]['pri
```

```
                    (point['type'] == 'trough' and point['price'] &lt; alternating[-1]['p
                    alternating[-1] = point

        return alternating

    def identify_crash_periods(self, turning_points: pd.DataFrame) -&gt; pd.DataFrame:
        """Identify crash periods (&gt;= 20% decline)."""
        crashes = []

        for i in range(len(turning_points) - 1):
            if (turning_points.iloc[i]['type'] == 'peak' and
                turning_points.iloc[i+1]['type'] == 'trough'):

                peak_price = turning_points.iloc[i]['price']
                trough_price = turning_points.iloc[i+1]['price']
                drawdown = (trough_price - peak_price) / peak_price

                if drawdown &lt;= -0.20:
                    crashes.append({
                        'start_date': turning_points.iloc[i]['date'],
                        'trough_date': turning_points.iloc[i+1]['date'],
                        'peak_price': peak_price,
                        'trough_price': trough_price,
                        'max_drawdown': drawdown
                    })

        return pd.DataFrame(crashes)
```

Phase 2 Acceptance Criteria

- [ ] All 28 indicators calculate correctly
- [ ] Regime detection identifies 8+ historical crashes
- [ ] Feature correlation analysis complete
- [ ] No redundant features (correlation < 0.95)
- [ ] Feature pipeline processes 55 years in < 5 minutes
- [ ] Complete documentation with formulas

PHASE 3: CRASH PREDICTION MODELS

Duration: 2 weeks
Dependencies: Phase 2 complete

Your Tasks

Build 5 models and combine into ensemble:

3.1 SVM Model

File: `src/models/crash_prediction/svm_model.py`

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
import numpy as np
import joblib

class SVMCrashPredictor:
    """SVM model for crash prediction."""

    def __init__(self):
        self.model = None
        self.scaler = StandardScaler()

    def train(self, X: np.ndarray, y: np.ndarray):
        """Train SVM with hyperparameter tuning."""
        X_scaled = self.scaler.fit_transform(X)

        param_grid = {
            'C': [0.1, 1, 10],
            'gamma': ['scale', 'auto'],
            'kernel': ['rbf']
        }

        self.model = GridSearchCV(
            SVC(probability=True, random_state=42),
            param_grid,
            cv=5,
            scoring='roc_auc'
        )

        self.model.fit(X_scaled, y)

    def predict_proba(self, X: np.ndarray) -&gt; np.ndarray:
        """Predict crash probability."""
        X_scaled = self.scaler.transform(X)
        return self.model.predict_proba(X_scaled)[:, 1]

    def save(self, filepath: str):
        """Save model."""
        joblib.dump({'model': self.model, 'scaler': self.scaler}, filepath)
```

3.2 Ensemble Model

File: `src/models/crash_prediction/ensemble_model.py`

```
import numpy as np
from typing import Dict, List

class CrashEnsemble:
    """Ensemble crash prediction model."""
```

```
    def __init__(self):
        self.models = {}
        self.weights = {}

    def add_model(self, name: str, model, weight: float = 1.0):
        """Add model to ensemble."""
        self.models[name] = model
        self.weights[name] = weight

    def calculate_optimal_weights(self, X_val: np.ndarray, y_val: np.ndarray):
        """Calculate weights based on validation AUC."""
        from sklearn.metrics import roc_auc_score

        aucs = {}
        for name, model in self.models.items():
            pred = model.predict_proba(X_val)
            aucs[name] = roc_auc_score(y_val, pred)

        total_auc = sum(aucs.values())
        self.weights = {name: auc/total_auc for name, auc in aucs.items()}

    def predict_proba(self, X: np.ndarray) -&gt; np.ndarray:
        """Ensemble prediction with weighted voting."""
        predictions = np.zeros(len(X))

        for name, model in self.models.items():
            pred = model.predict_proba(X)
            predictions += self.weights[name] * pred

        return predictions
```

Phase 3 Acceptance Criteria

- [ ] Ensemble AUC >= 0.85 on out-of-sample tests
- [ ] Overall accuracy >= 85%
- [ ] False alarm rate <= 20%
- [ ] Recall >= 75%
- [ ] Validated on 2008, 2020, 2022 crashes
- [ ] Walk-forward validation complete
- [ ] Models saved with versioning

PHASE 4: BOTTOM PREDICTION MODELS

Duration: 2 weeks
Dependencies: Phase 3 complete

Your Tasks

4.1 MLP Model

File: `src/models/bottom_prediction/mlp_model.py`

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler

class MLPBottomPredictor:
    """MLP for predicting market bottoms."""

    def __init__(self, input_dim: int = 15):
        self.input_dim = input_dim
        self.model = None
        self.scaler = StandardScaler()

    def build_model(self) -&gt; Sequential:
        """Build 32-16-8-1 architecture."""
        model = Sequential([
            Dense(32, activation='relu', input_shape=(self.input_dim,)),
            BatchNormalization(),
            Dropout(0.3),

            Dense(16, activation='relu'),
            BatchNormalization(),
            Dropout(0.2),

            Dense(8, activation='relu'),
            Dropout(0.1),

            Dense(1, activation='linear')
        ])

        model.compile(optimizer=Adam(0.001), loss='mse', metrics=['mae'])
        return model

    def train(self, X: np.ndarray, y: np.ndarray):
        """Train model."""
        X_scaled = self.scaler.fit_transform(X)
        self.model = self.build_model()

        history = self.model.fit(
            X_scaled, y,
            epochs=200,
            batch_size=32,
            validation_split=0.2,
            verbose=1
        )

        return history
```

```
    def predict(self, X: np.ndarray) -&gt; np.ndarray:
        """Predict days to bottom."""
        X_scaled = self.scaler.transform(X)
        return self.model.predict(X_scaled).flatten()
```

Phase 4 Acceptance Criteria

- [ ] Bottom prediction 85-90% accurate (±30 days)
- [ ] Recovery time prediction within ±3 months
- [ ] Validated on 8+ crashes
- [ ] 2/3 exact bottom predictions
- [ ] MLP MSE <= 0.05
- [ ] 90% confidence intervals calibrated

PHASE 5: DASHBOARD DEVELOPMENT

Duration: 1-2 weeks
Dependencies: Phase 4 complete

Your Tasks

5.1 Main App

File: `src/dashboard/app.py`

```
import streamlit as st
import pandas as pd
import plotly.graph_objects as go

st.set_page_config(page_title="Market Crash Predictor", layout="wide")

st.title("Market Crash &amp;amp; Bottom Prediction System")

# Sidebar navigation<a></a>
page = st.sidebar.selectbox("Navigate", ["Home", "Indicators", "Predictions", "Settings"]

if page == "Home":
    st.header("System Overview")

    col1, col2, col3 = st.columns(3)

    with col1:
        st.metric("Crash Probability", "23%", "-5%")

    with col2:
        st.metric("Days to Bottom", "N/A", "")

    with col3:
        st.metric("Model Confidence", "High", "")
```

```
elif page == "Indicators":
    st.header("28 Crash Indicators")

    # Display each indicator with chart
    for indicator in range(1, 29):
        with st.expander(f"Indicator {indicator}"):
            st.line_chart([1,2,3,4,5])

elif page == "Predictions":
    st.header("Model Predictions")

    # Crash probability gauge
    fig = go.Figure(go.Indicator(
        mode="gauge+number",
        value=23,
        title={'text': "Crash Probability (%)"},
        gauge={'axis': {'range': [0, 100]},
               'bar': {'color': "red"}}
    ))
    st.plotly_chart(fig)

elif page == "Settings":
    st.header("Alert Configuration")

    st.slider("Crash Probability Threshold", 0, 100, 50)
    st.text_input("Email Address")
    st.button("Save Settings")
```

5.2 Individual Indicator Charts

File: `src/dashboard/pages/indicators.py`

Create individual charts for ALL 28 indicators showing:

- Historical trend (5+ years)
- Current value with color coding
- Threshold levels
- Explanation text

Phase 5 Acceptance Criteria

- [ ] Dashboard loads in < 2 seconds
- [ ] All 28 indicators show individual charts
- [ ] Crash probability gauge works
- [ ] Bottom prediction timeline displays
- [ ] Threshold sliders functional
- [ ] Users can configure alerts
- [ ] Responsive design (desktop + mobile)

PHASE 6: ALERT SYSTEM

Duration: 1 week
Dependencies: Phase 5 complete

Your Tasks

6.1 Email Alerts

File: `src/alerts/email_alert.py`

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import logging

class EmailAlert:
    """Send email alerts via SMTP."""

    def __init__(self, smtp_server: str, smtp_port: int, username: str, password: str):
        self.smtp_server = smtp_server
        self.smtp_port = smtp_port
        self.username = username
        self.password = password
        self.logger = logging.getLogger(__name__)

    def send_alert(self, to_email: str, subject: str, body: str) -&gt; bool:
        """Send email alert."""
        try:
            msg = MIMEMultipart()
            msg['From'] = self.username
            msg['To'] = to_email
            msg['Subject'] = subject

            msg.attach(MIMEText(body, 'plain'))

            server = smtplib.SMTP(self.smtp_server, self.smtp_port)
            server.starttls()
            server.login(self.username, self.password)
            server.send_message(msg)
            server.quit()

            self.logger.info(f"Email sent to {to_email}")
            return True

        except Exception as e:
            self.logger.error(f"Email failed: {e}")
            return False
```

6.2 Alert Manager

File: `src/alerts/alert_manager.py`

```
from typing import Dict, List
import logging

class AlertManager:
    """Manages alert conditions and delivery."""

    def __init__(self):
        self.conditions = []
        self.email_alert = None
        self.sms_alert = None
        self.logger = logging.getLogger(__name__)

    def add_condition(self, name: str, check_func, threshold: float):
        """Add alert condition."""
        self.conditions.append({
            'name': name,
            'check': check_func,
            'threshold': threshold
        })

    def evaluate_conditions(self, data: Dict) -&gt; List[str]:
        """Evaluate all conditions and return triggered alerts."""
        triggered = []

        for condition in self.conditions:
            if condition['check'](data, condition['threshold']):
                triggered.append(condition['name'])
                self.logger.info(f"Alert triggered: {condition['name']}")

        return triggered

    def send_alerts(self, triggered: List[str], recipients: List[str]):
        """Send alerts for triggered conditions."""
        for alert_name in triggered:
            subject = f"Market Alert: {alert_name}"
            body = f"Alert condition '{alert_name}' has been triggered."

            for email in recipients:
                if self.email_alert:
                    self.email_alert.send_alert(email, subject, body)
```

Phase 6 Acceptance Criteria

- [ ] Email alerts deliver within 60 seconds
- [ ] SMS alerts deliver within 120 seconds
- [ ] Alert delivery >= 99%
- [ ] No duplicate alerts within 24 hours
- [ ] Alerts work for indicators AND models

- [ ] Configuration persists

PHASE 7: INTEGRATION & TESTING

Duration: 1 week
Dependencies: All phases complete

Your Tasks

7.1 Integration Tests

File: `tests/test_integration.py`

```
import pytest
from src.data_collection.fred_collector import FREDCollector
from src.feature_engineering.crash_indicators import CrashIndicators
from src.models.crash_prediction.ensemble_model import CrashEnsemble

def test_end_to_end_pipeline():
    """Test complete data flow."""
    # Collect data
    collector = FREDCollector(api_key="test")
    # Process features
    # Run models
    # Generate predictions
    assert True

def test_daily_pipeline_runs():
    """Test daily pipeline executes without errors."""
    # Run scheduler once
    assert True
```

7.2 Performance Testing

Verify:

- Data collection completes in < 5 minutes
- Feature generation in < 2 minutes
- Model inference in < 1 minute
- Dashboard loads in < 2 seconds

7.3 Documentation

Create:

- README.md with setup instructions
- USER_GUIDE.md with screenshots
- API_DOCUMENTATION.md with all functions

- MODEL_METHODOLOGY.md with math and results

Phase 7 Acceptance Criteria

- [ ] All integration tests pass
- [ ] System runs 7 days without errors
- [ ] Daily pipeline < 10 minutes total
- [ ] Dashboard responds in < 2 seconds
- [ ] Error recovery tested
- [ ] Complete documentation
- [ ] Deployment successful

FINAL CHECKLIST

Before marking complete:

- [ ] Crash prediction AUC >= 0.85
- [ ] Crash prediction accuracy >= 85%
- [ ] Bottom prediction 85-90% accurate
- [ ] All 28 indicators display with charts
- [ ] Users can set thresholds per indicator
- [ ] Alerts work for indicators AND models
- [ ] Test coverage >= 85%
- [ ] System runs 7 days error-free
- [ ] Documentation complete
- [ ] README has setup instructions

IMPLEMENTATION NOTES

Code Quality Standards

Every file must have:

- Type hints on all functions
- Docstrings in Google format
- Comprehensive error handling
- Unit tests (>= 85% coverage)
- Logging statements
- No hardcoded values

Example:

```
def calculate_yield_spread(df: pd.DataFrame) -&gt; pd.Series:
    """
    Calculate 10Y-3M Treasury yield spread.

    Args:
        df: DataFrame with yield_10y and yield_3m columns

    Returns:
        Series with yield spread values

    Raises:
        ValueError: If required columns missing
    """
    if 'yield_10y' not in df.columns:
        raise ValueError("Missing yield_10y column")

    return df['yield_10y'] - df['yield_3m']
```

Testing Requirements

Every module needs:

- Unit tests for all functions
- Integration tests for workflows
- Edge case tests
- Mock external APIs

Run: `pytest --cov=src tests/`

Target: >= 85% coverage

Dependencies

Install all from requirements.txt:

```
fredapi&gt;=0.5.1
yfinance&gt;=0.2.41
alpha-vantage&gt;=2.3.1
pandas&gt;=2.2.0
numpy&gt;=1.26.0
scikit-learn&gt;=1.4.0
tensorflow&gt;=2.15.0
xgboost&gt;=2.0.0
streamlit&gt;=1.31.0
plotly&gt;=5.18.0
twilio&gt;=8.11.0
APScheduler&gt;=3.10.4
sqlalchemy&gt;=2.0.25
pytest&gt;=8.0.0
pytest-cov&gt;=4.1.0
```

```
black&gt;=24.1.0
pylint&gt;=3.0.0
```

You now have complete specifications for all 7 phases. Implement sequentially, starting with Phase 1. Each phase must pass acceptance criteria before proceeding to the next.

Good luck building this production-ready system!