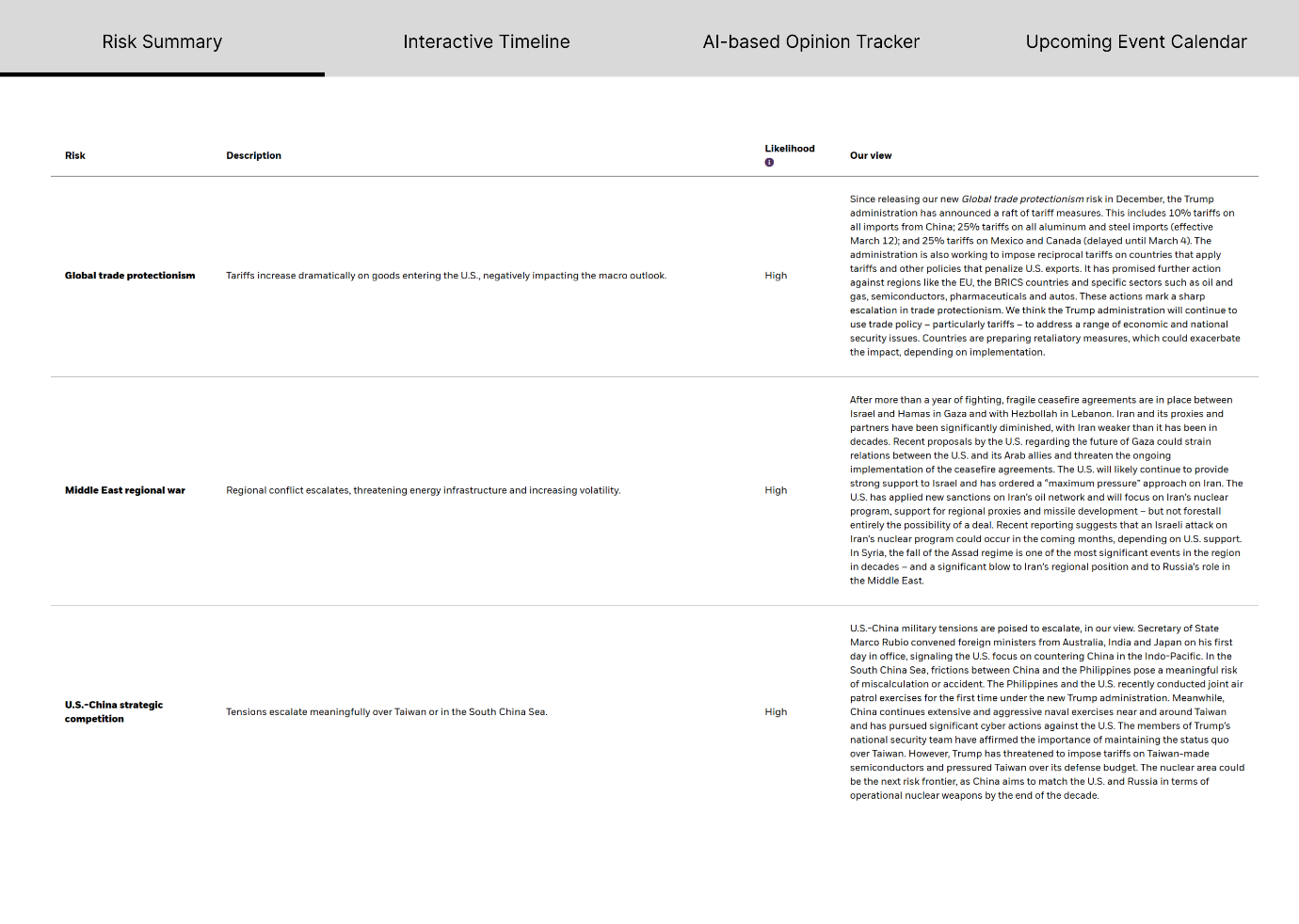
**Sprint Progress Report 1**

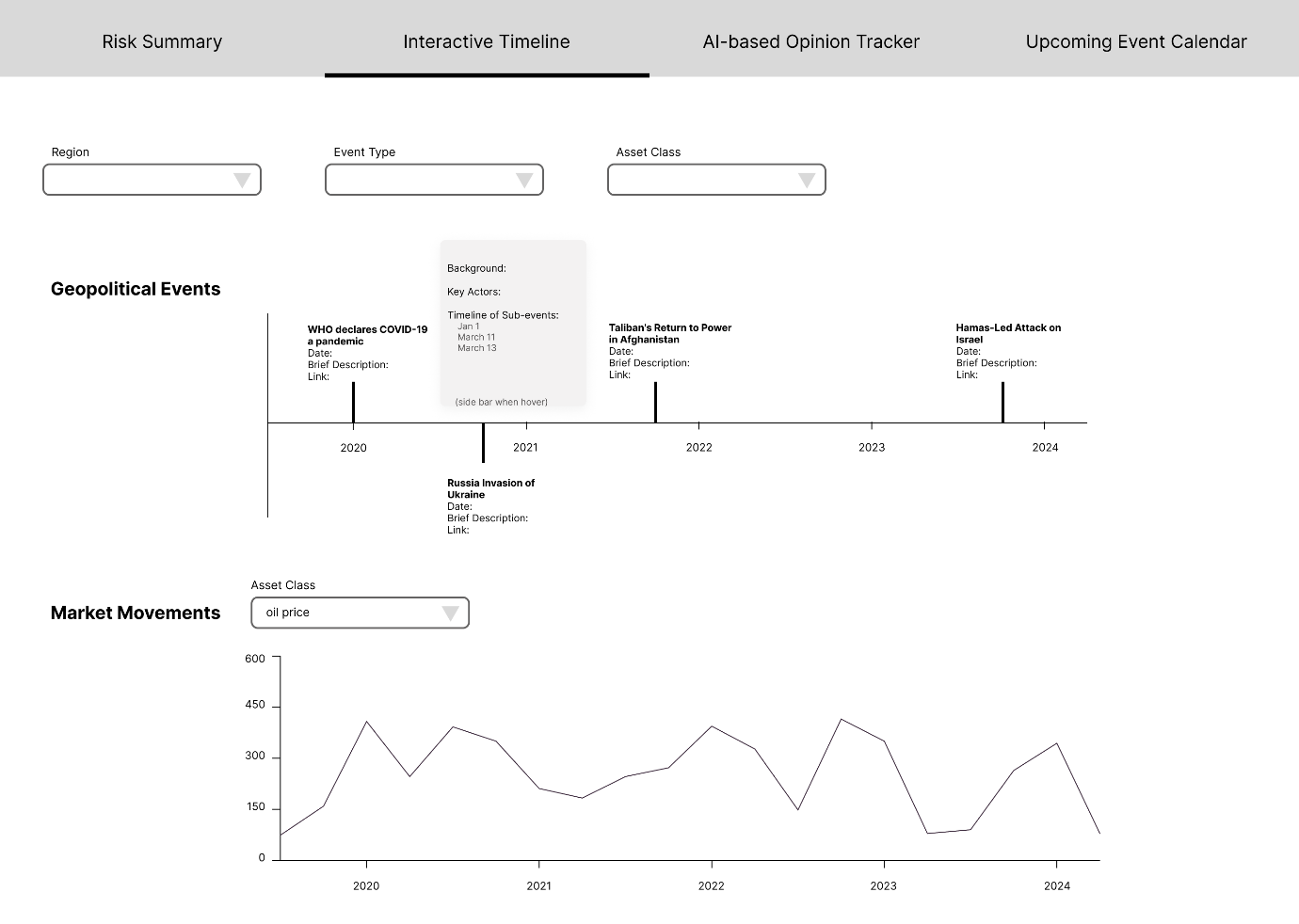
1. **Schedule the six sprint meetings with your assigned TA.**
2. **Create a lean canvas for your project.**

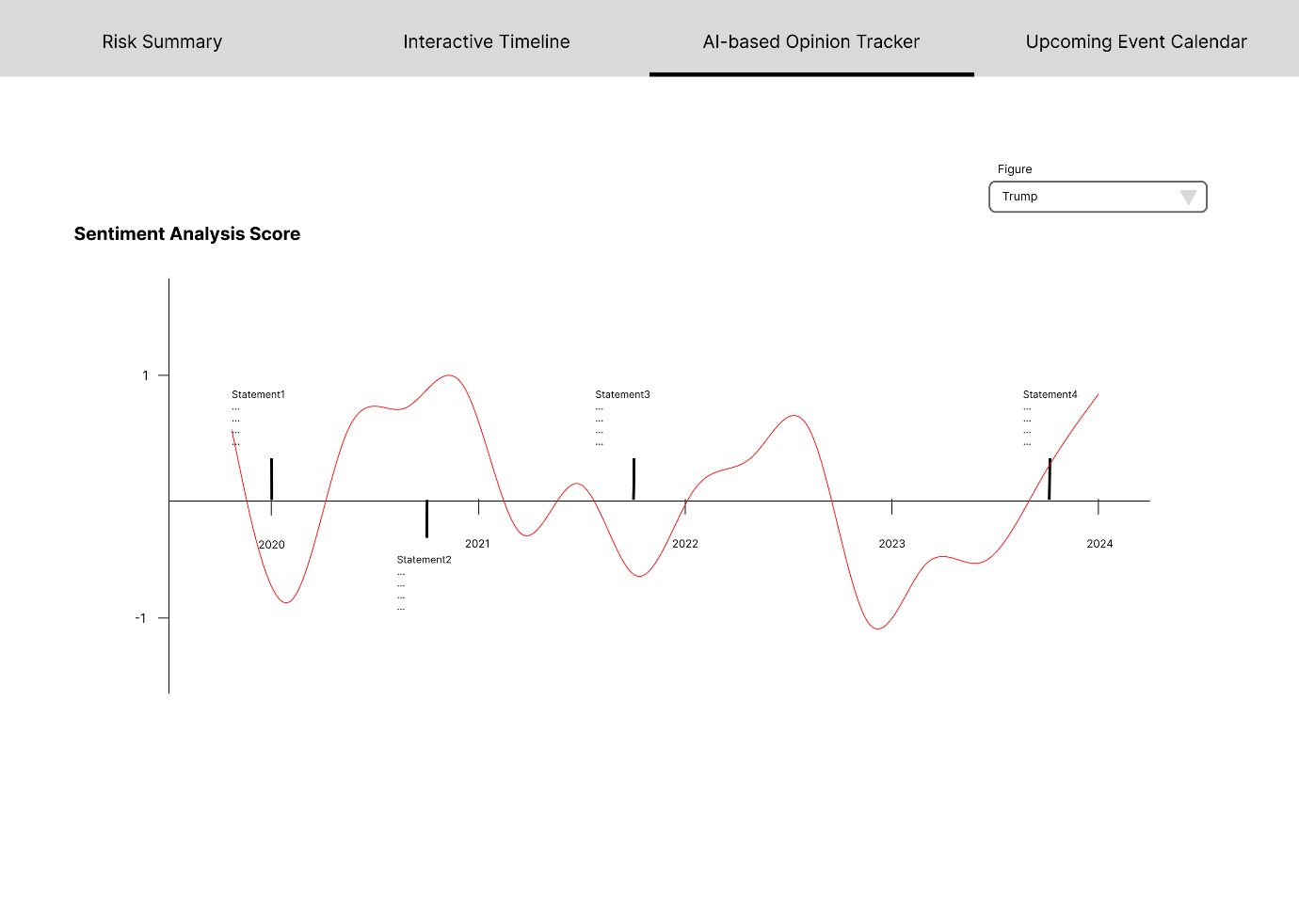
**A screenshot of a computer

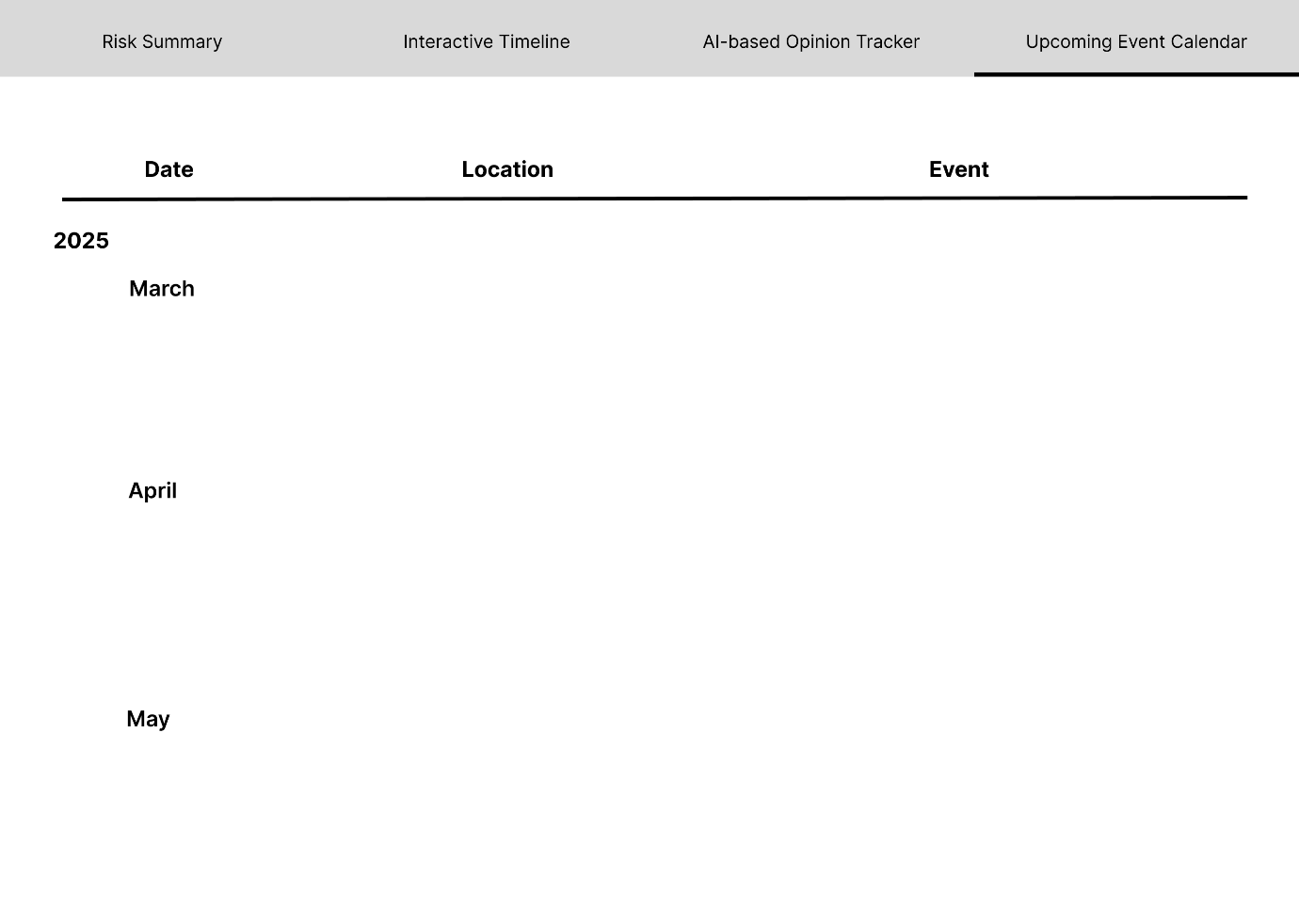
AI-generated content may be incorrect.**

1. **Develop two or more paper prototype screens to highlight the significant functionality of your application.**









1. **Present your project proposal (Lean Canvas and paper prototypes) during a five-minute session in class.  (Section 02 only)**
2. **Establish your Gitlab repository.**

https://gitlab.oit.duke.edu/xl477/fintech-512-geopolitical-dashboard.git

1. **Based upon the overall project requirements, establish the specific requirements for your project.**

* Requirement 1: Risk Summary
* Requirement 2: Interactive Timeline
* Requirement 3: Opinion Tracker UI & Timeline Integration
* Requirement 4: Upcoming Events Calendar

1. **Based upon those requirements and items specified in this document, create a “product backlog”. Use GitLab issues to store your project backlog.**

In GitLab issues

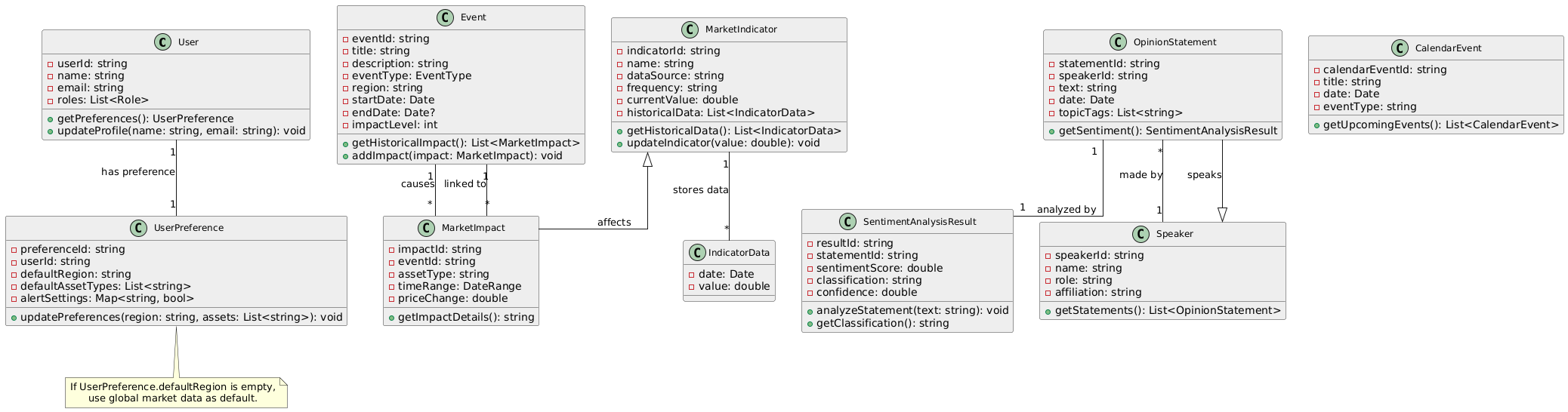
1. **Perform the initial design for your application. Items to consider:**

* **Architecture**

A diagram of a computer

AI-generated content may be incorrect.

* **Class design**



**1. User**

The User class represents an individual using the geopolitical risk dashboard, such as a portfolio manager or analyst. It stores personal details and system roles, allowing users to manage their profiles and preferences.

* **Attributes:**
  + userId: Unique identifier for the user.
  + name: Full name of the user.
  + email: Contact email address.
  + roles: A list of assigned system roles.
* **Methods:**
  + getPreferences(): Retrieves the user's saved preferences.
  + updateProfile(name, email): Updates user information, such as name and email.

**2. UserPreference**

The UserPreference class manages personalized settings for each user. It allows customization of the dashboard by selecting preferred regions, asset types, and alert configurations.

* **Attributes:**
  + preferenceId: Unique identifier for the preference record.
  + userId: The ID of the associated user.
  + defaultRegion: Preferred geopolitical region to track.
  + defaultAssetTypes: List of selected asset categories (e.g., oil, FX).
  + alertSettings: A dictionary storing user-defined alert preferences.
* **Methods:**
  + updatePreferences(region, assets): Modifies the user's selected regions and asset classes.

**3. Event**

The Event class represents significant geopolitical developments that may impact financial markets. It contains essential details about each event, such as its type, affected region, and impact level.

* **Attributes:**
  + eventId: Unique identifier for the event.
  + title: Name or headline of the event.
  + description: A brief summary of the event.
  + eventType: Category of the event (e.g., election, war, policy change).
  + region: The geographical location affected by the event.
  + startDate: The event's starting date.
  + endDate: The event's ending date (if applicable).
  + impactLevel: Estimated effect on financial markets.
* **Methods:**
  + getHistoricalImpact(): Retrieves past market impacts related to the event.
  + addImpact(impact): Links a new market impact to the event.

**4. MarketImpact**

The MarketImpact class tracks how geopolitical events influence financial assets such as commodities, currencies, and stock indices.

* **Attributes:**
  + impactId: Unique identifier for the impact entry.
  + eventId: The event associated with this impact.
  + assetType: The type of affected asset (e.g., crude oil, USD).
  + timeRange: The time period over which the impact was measured.
  + priceChange: The observed price fluctuation due to the event.
* **Methods:**
  + getImpactDetails(): Returns a summary of the asset's market reaction.

**5. MarketIndicator**

The MarketIndicator class represents financial metrics used to assess geopolitical risks. It includes data such as commodity prices, volatility indices, and geopolitical risk scores.

* **Attributes:**
  + indicatorId: Unique identifier for the indicator.
  + name: The name of the market indicator.
  + dataSource: The source providing the indicator data (e.g., Refinitiv, EIA).
  + frequency: The data update interval (e.g., daily, hourly).
  + currentValue: The most recent recorded value.
  + historicalData: A list of past values for the indicator.
* **Methods:**
  + getHistoricalData(): Retrieves historical time-series data.
  + updateIndicator(value): Updates the current value of the indicator.

**6. IndicatorData**

The IndicatorData class stores historical records of a market indicator's values over time.

* **Attributes:**
  + date: The date of the recorded value.
  + value: The numerical value of the indicator at that time.

**7. OpinionStatement**

The OpinionStatement class records statements from influential figures such as policymakers, central bankers, and industry executives. These statements can be analyzed for sentiment and geopolitical significance.

* **Attributes:**
  + statementId: Unique identifier for the statement.
  + speakerId: The person who made the statement.
  + text: The full text of the statement.
  + date: The date the statement was issued.
  + topicTags: Keywords categorizing the statement.
* **Methods:**
  + getSentiment(): Retrieves the sentiment analysis result for the statement.

**8. SentimentAnalysisResult**

The SentimentAnalysisResult class stores the results of AI-driven sentiment analysis on public statements. It helps assess whether a statement is positive, negative, or neutral toward a given market or policy.

* **Attributes:**
  + resultId: Unique identifier for the sentiment analysis result.
  + statementId: The statement that was analyzed.
  + sentimentScore: A numerical representation of sentiment.
  + classification: Sentiment category (e.g., hawkish, dovish, neutral).
  + confidence: Confidence level of the classification.
* **Methods:**
  + analyzeStatement(text): Processes a new statement for sentiment analysis.
  + getClassification(): Returns the sentiment classification.

**9. Speaker**

The Speaker class represents individuals who issue public statements that may impact markets, such as government officials, corporate executives, or central bankers.

* **Attributes:**
  + speakerId: Unique identifier for the speaker.
  + name: Full name of the speaker.
  + role: Their title or position (e.g., Federal Reserve Chair).
  + affiliation: The organization they are associated with.
* **Methods:**
  + getStatements(): Retrieves all statements made by the speaker.

**10. CalendarEvent**

The CalendarEvent class tracks upcoming geopolitical and economic events that may impact financial markets. Examples include central bank meetings, elections, and OPEC summits.

* **Attributes:**
  + calendarEventId: Unique identifier for the calendar event.
  + title: Event name or description.
  + date: The scheduled date of the event.
  + eventType: The type of event (e.g., policy announcement, trade summit).
* **Methods:**
  + getUpcomingEvents(): Retrieves a list of future scheduled events.

**11. RiskScore**

The RiskScore class calculates and stores geopolitical risk levels based on various market and event factors.

* **Attributes:**
  + scoreId: Unique identifier for the risk score.
  + date: The date the score was calculated.
  + region: The geographic area assessed.
  + assetClass: The financial asset affected (e.g., FX, commodities).
  + value: The computed risk score.
  + methodology: The approach used to determine the score.
* **Methods:**
  + calculateRisk(): Computes the risk score based on event data and market conditions.
  + getRiskLevel(): Returns a risk assessment in qualitative terms (e.g., low, moderate, high).
* **Database design**

**Events Table**

|  |  |
| --- | --- |
| Field | Description |
| Event ID | Unique identifier |
| Event Name | Name of the event |
| Date | Date of occurrence |
| Description | Detailed description |
| Region | Geographic area |
| Event Type | Category (e.g., war, sanction) |
| Impact Level | High, Medium, Low |
| Links | Sources or news articles |

**Opinions Table**

|  |  |
| --- | --- |
| Field | Description |
| Figure ID | Unique identifier |
| Name | Name of the figure |
| Position | Role (e.g., President) |
| Country/Org | Affiliation |
| Statement | Text of the statement |
| Date | Date of statement |
| Source | Link to source (e.g., X post) |
| Related Event | Linked event ID |

* **APIs**

Financial data: Refinitiv, FactSet

Energy data/reports: IEA

News: Refinitiv, FactSet, Stratr, Jane, RAND

Key Figure’s Statement: Government official websites

1. https://www.state.gov/press-releases/

2. https://ustr.gov/about-us/policy-offices/press-office/press-releases

3. https://appropriations.house.gov/media-original

4. https://www.congress.gov/congressional-record

5. https://www.defense.gov/News

6. https://ofac.treasury.gov/recent-actions

7. https://home.treasury.gov/news/press-releases

8. https://www.commerce.gov/news?

* **Testing approach**

Basically use pytest as testing approach.

**1. Test Organization and Structure**

* **Directory Layout**:
  + Separate tests by context (e.g., for isolated logic, for inter-service checks).
  + Maintain a shared to manage reusable fixtures like database sessions or common test data.
* **Naming Conventions**:
  + Follow a clear naming scheme (e.g., ), and use descriptive test function names (e.g., ).

**2. Unit Testing with Pytest**

* **Primary Goal**: Validate individual components in isolation, such as event services, risk calculation modules, or data-cleaning utilities.
* **Fixtures**:
  + Use fixtures to set up and tear down any ephemeral dependencies (e.g., a mock object or a temporary configuration).
  + Keep fixtures at the smallest necessary scope (function, module, or session) so that each test remains self-contained.
* **Mocking**:
  + Mock external calls (e.g., network requests to EIA or Refinitiv) to avoid brittle tests that rely on external APIs being up.
  + Mock time-based or random functionality to ensure repeatability.
* **Assertions**:
  + Rely on plain statements to keep tests readable.
  + Consider specialized assertion libraries for more descriptive error reporting (e.g., for exception checks).

**3. Integration Testing**

* **Goal**: Ensure services operate correctly when communicating with each other or with shared resources (e.g., databases, caches).
* **Database Validation**:
  + Use an in-memory or temporary database to confirm that operations like creating events, storing sentiment analysis results, or retrieving market indicators work end-to-end.
  + Validate constraints and relationships (e.g., checking that an event must have a region before saving).
* **API Layer**:
  + Spin up local instances of the relevant microservices (or a monolithic backend) and call them with test data to verify the correctness of endpoints.
  + Ensure that service orchestration—for instance, an event insertion triggering risk calculation—works as intended.
* **Shared Fixtures**:
  + Maintain common fixtures in for integration tests, such as a fixture that creates and tears down a test database schema.

**4. Data-Oriented Testing**

* **Data Quality**:
  + Validate that ingestion logic handles malformed or missing fields (e.g., incomplete JSON).
  + Use parameterized tests for boundary cases like extremely high or negative price values.
* **Transformation Checks**:
  + Confirm that data from external APIs is normalized to the system’s internal format (e.g., date/time zones, numeric formatting).
  + Verify that cleaning steps (e.g., removing duplicates or out-of-range data) are correctly applied.

**5. Pytest Features to Leverage**

* **Parameterization**:
  + Cover multiple input-output scenarios for complex logic (e.g., sentiment thresholds, risk score boundaries) without duplicating test code.
* **Markers**:
  + Use markers (e.g., , ) to selectively run or skip certain test suites.
  + Organize slow or specialized tests separately, ensuring quick feedback on core functionality.
* **Fixtures Scope**:
  + Strategically define fixture scope to balance performance with isolation. For example, a -scoped fixture for setting up a test database versus a -scoped fixture for ephemeral mock objects.

**6. End-to-End (E2E) Scenarios**

* **System Workflow**:
  + Test complete flows, from data ingestion through to displaying risk scores on the dashboard.
  + Focus on real-world user paths—e.g., “user logs in, configures a region filter, sees relevant events, and sets an alert.”
* **No UI Code Needed**:
  + You can still use Pytest to perform E2E checks with HTTP requests if the frontend and backend are decoupled.
  + Alternatively, integrate with a browser automation tool (like Selenium or Playwright) but keep orchestration in Pytest for consistency.

**7. Performance Considerations**

* **Load Testing**:
  + While Pytest isn’t primarily a load-testing tool, you can still create stress tests that loop over multiple parallel requests to a local test environment.
  + Combine with Pytest markers to differentiate standard unit tests from heavier performance scenarios.
* **Caching & Timeouts**:
  + Validate that the system gracefully handles slower or temporarily unavailable data sources (e.g., network failures).

**8. Reporting and Coverage**

* **Coverage**:
  + Use coverage plugins within Pytest to measure how much of the system’s logic is tested.
  + Identify untested lines or branches, especially in critical modules like risk calculation or data ingestion.
* **Reporting**:
  + Generate HTML or XML reports for continuous integration systems, providing insights into pass rates, coverage trends, and potential regressions.

**9. Maintenance and Best Practices**

* **Refactor Tests**:
  + Keep test code clean and maintainable—avoid overly complex test setups or deep nesting in fixture dependencies.
  + Regularly remove deprecated tests if the functionality no longer exists.
* **Incremental Improvements**:
  + As new data sources or event types are added, extend existing tests or create new ones to ensure coverage.
  + Encourage a “test-first” or at least “test-adjacent” mindset to reduce technical debt.
* **Error handling**

1.1 Data Ingestion & Processing Errors

* **Potential Issues:**
  + Malformed or incomplete JSON from external APIs (e.g., missing fields in an event).
  + Network or timeout errors when pulling data from Refinitiv, FactSet, or EIA.
  + Authentication/authorization failures (invalid API keys).
* **Proposed Handling:**
  + **Validation**: Parse incoming data against a strict schema; reject or flag incomplete records.
  + **Retries & Backoff**: Implement exponential backoff for transient errors (network or API rate limits).
  + **Logging & Alerts**: Log all ingestion failures with timestamps; send alerts if critical feeds fail repeatedly.
* **Representative Example**
  + If a call to Refinitiv returns empty data:
    1. Log error: “Refinitiv data fetch failed: empty response.”
    2. Retry after 30 seconds, then 2 minutes, etc.
    3. If still failing, raise an alert to the engineering team.

1.2 Backend Service & Database Errors

* **Potential Issues:**
  + Database connection loss or migration conflict.
  + Transaction rollbacks due to constraint violations (e.g., foreign key constraints in event logs).
  + Computation errors in business logic (e.g., division by zero in risk scoring).
* **Proposed Handling:**
  + **Connection Pooling**: Auto-reconnect logic for transient connection issues.
  + **Transaction Boundaries**: Wrap operations in transactions; rollback on exceptions.
  + **Graceful Degradation**: If a calculation fails, return a fallback (“Risk Score Unavailable”).
* **Representative Example**
  + If saving a new “Event” record violates a database constraint:
    - Catch IntegrityError, log details, return an error message: “Event creation failed due to invalid data.”

1.3 Frontend & UX Errors

* **Potential Issues:**
  + API endpoint unreachable or responds with unexpected status code.
  + Visualization library (e.g., timeline or chart) throws runtime error due to mismatched data format.
* **Proposed Handling:**
  + **User-Friendly Messages**: Show non-technical error notifications (e.g., “Unable to load events, please try again”).
  + **Fallback UI**: If the timeline fails, display partial data or hide the component with a retry option.
  + **Logging**: Capture frontend errors (e.g., using browser-based error tracking tools like Sentry) for investigation.
* **Representative Example**
  + If the timeline service is offline:
    - Hide timeline widget; show a notice: “Timeline data temporarily unavailable. Try refreshing in a few minutes.”

1.4 Security & Authorization Errors

* **Potential Issues:**
  + Unauthorized access attempts to restricted endpoints or admin features.
  + Expired or invalid JWT tokens.
* **Proposed Handling:**
  + **RBAC/ABAC**: Validate user roles at each endpoint.
  + **Graceful Redirect**: If token is invalid or expired, redirect to login, preserving user state.
  + **Auditing**: Log unauthorized attempts with IP and user ID (if any).
* **Representative Example**
  + A user tries to access the “Scenario Simulation” feature without the right role:
    - Backend returns HTTP 403 (Forbidden).
    - Frontend shows: “You do not have permission to access this feature.”

1. **Implementing necessary non-functional requirements (security/performance)**

Below is a **table** of key NFRs, each with a **brief explanation** and example solution:

| **NFR Category** | **Requirement** | **Example Solution** |
| --- | --- | --- |
| **Performance** | **Low latency** (<500ms API response) | - Use efficient DB indexing <br/> - Employ caching (Redis) for frequent queries |
|  | **Concurrent users** (≥1000) | - Horizontal scaling with Kubernetes <br/> - Load balancers (NGINX/HAProxy) |
| **Reliability** | **High availability** (≥99.9%) | - Deploy across multiple availability zones <br/> - Automated health checks |
|  | **Recovery** (RPO < 1 hour) | - Frequent DB backups <br/> - Automatic failover to secondary database |
| **Scalability** | Easy to add **new data sources** or extra features | - Microservices architecture <br/> - Well-documented API endpoints |
| **Maintainability** | **Clear code structure** + minimized technical debt | - Follow consistent naming & structure <br/> - Code reviews & regular refactoring |
| **Security** | **Data privacy** (GDPR compliance) | - Encrypt data in transit and at rest <br/> - Consent management |
|  | **Access control** (RBAC) | - Role-based permissions <br/> - Secure auth tokens |
| **Usability** | **User-friendly UI** with minimal learning curve | - Provide in-app tutorials <br/> - Keep designs consistent & intuitive |
| **Extensibility** | **Integration** with external analytics tools or custom scripts | - Provide REST APIs with documentation <br/> - Maintain stable versioning |

3. Potential Solution & Roadmap

Below is a **phased** roadmap, showing how you can roll out the dashboard with increasing functionality and robustness:

Phase 1: MVP

* **Core Components**:
  1. Basic event ingestion (timeline).
  2. Database setup for storing events, opinions, indicators.
  3. Simple UI to display key geopolitical events with minimal filtering.
* **Representative Example**:
  1. Launch a minimal timeline page showing the last 30 days of significant events (e.g., OPEC meeting, major election).

Phase 2: Data & Analytics Expansion

* **Enhancements**:
  1. Integrate real-time market feeds (oil, gas, FX) from Refinitiv/FactSet.
  2. Implement sentiment analysis with LLM-based engine (e.g., Wiseflow or OpenAI).
  3. Add correlation charts between events and market movements.
* **Representative Example**:
  1. Show how Brent oil price reacts to major Middle East developments; highlight correlation on a chart.

Phase 3: Personalization & Alerts

* **Enhancements**:
  1. Allow user-specific filters (region, asset class).
  2. Implement alerts & notifications (e.g., text/email when conflict risk is elevated).
  3. Improve UI with refined data visualizations and interactive drill-downs.
* **Representative Example**:
  1. A portfolio manager sets up “High-impact events in Asia” alerts via SMS for immediate risk assessment.

Phase 4: Advanced Scenario Simulation

* **Enhancements**:
  1. Let users create hypothetical “What-if” scenarios (e.g., supply disruption).
  2. Provide modeling tools that estimate price/volatility responses.
  3. Export scenario data to Excel or other analytics platforms.
* **Representative Example**:
  1. Analyst simulates “10% cut in Saudi oil production for 3 months,” generating predicted price impacts in real time.

Ongoing: Maintenance & Optimization

* Monitor performance, fix bugs, refine user experience.
* Add new data feeds or risk indices (e.g., new geopolitical risk providers).
* Conduct periodic security and compliance audits.

Final Notes & Recommendations

1. **Centralized Logging and Monitoring**: Use tools like **ELK Stack** or **Datadog** for logs and metrics.
2. **Continuous Integration/Continuous Deployment (CI/CD)**: Automate testing and deployment (e.g., GitHub Actions, Jenkins).
3. **Regular Stakeholder Feedback**: Schedule user testing at each phase to refine features (particularly usability and personalization).
4. **Scalability Planning**: Design with future expansions in mind—modular microservices, containerized deployments, load balancing.

* **Outline your initial approach to incrementally building your application.**

Phase 1: Setting Up the Foundation

**Tasks:**

* Set up GitLab Repository for version control.
* Define database schema
* Identify reliable data sources (news, market data).
* Implement API connectivity to pull raw data from selected sources.

**Deliverables:**

* A working database structure with sample data.
* API connection established for news (geopolitical events) and financial data

Phase 2

1. **Risk summary & Upcoming event page (part1)**

**Objective: Build database, frontend UI to display news, but without the feature of AI-powered relevance scoring & alert and market movement charts**

**Tasks:**

* Integrate news APIs and develop data extraction scripts.
* Implement data cleaning and storage logic.
* Develop a basic frontend UI to display news data.
* Implement automated testing to verify API parsing and database storage.

**Deliverables:**

* A structured database with parsed and cleaned news.
* Working risk summary / upcoming event dashboard pages displaying news data.

1. **Interactive Timeline page (part1)**

**Objective: Geopolitical Timeline with Filters and Event Details**

**Tasks:**

* Develop the timeline UI with filtering capabilities.
* Connect it to the event database for data retrieval.
* Implement hover and click features to display event details.

**Deliverables:**

* A working timeline visualization with basic filters.

1. **AI-Based Opinion Tracker page (part1)**

**Objective: Build database, frontend UI to display the timeline, without building the feature of sentiment analysis & categorization**

**Tasks:**

* Develop a data pipeline for news transcripts, speeches, social media.
* Store structured statements in the database.

**Deliverables:**

* A structured database of key statements.

Phase 3

1. **Interactive Timeline page (part2)**

**Objective: Build Market Movement Charts for Financial Assets**

**Tasks:**

* + Integrate financial data sources (e.g., Refinitiv, FactSet).
  + Implement market movement visualization (e.g., oil prices, stock indices).
  + Enable data overlays and hover insights.

**Deliverables:**

* + Market movement charts displaying financial data.

1. **Risk summary page (part2)**

**Objective: Implement an AI model to rank news articles based on relevance + Build an alert feature**

**Tasks:**

* Train a basic ML model to assign relevance scores.
* Add a search function and custom alerts.
* Validate news ranking accuracy with test cases.

**Deliverables:**

* A dynamic Risk Summary that ranks relevant news.
* A working search and alert system for users.

1. **AI-Based Opinion Tracker page (part 2 – Train a sentiment analysis & categorization ML model)**

**Objective:** Implement AI-powered sentiment analysis to classify statements.

**Tasks:**

* Train and integrate NLP models for sentiment classification.
* Categorize statements (e.g., "hawkish," "dovish," "neutral").

**Deliverables:**

* A sentiment-labeled dataset for key financial statements.

Phase 4

**AI-Based Opinion Tracker page (part 3)**

**Objective: Use the sentiment analysis & categorization ML model from Phase 3 to display the timeline**

**Tasks:**

* Link sentiment data to the interactive timeline.
* Enable filtering by speaker and date range.

**Deliverables:**

* A sentiment-powered interactive timeline.

Phase 5: Integration, Testing, and Deployment

* Final integration of all increments
* Automated testing for core features
* User acceptance testing (UAT)
* Bug fixes and optimizations
* Deploy application in production environment.

1. **Hold your first weekly sprint meeting.**
2. **Complete and submit your first sprint status report.**