Mental Health Trends from Social Media (Reddit) Data

Applied Data Science, San Jose State University
Data 228: Big Data Project
Group: 09
Submitted By: Chandana Rondla, Deeksha Chauhan, Himaja Sree Thota, Moksh Aggarwal, Saurabh Suman, Shivam Mahendru
Submitted To: Professor Sangjin Lee
May 10, 2025

Table of Contents

1. Introduction & Motivation

In an era of hyper-connectivity and relentless change, stress-related disorders are outpacing traditional public-health monitoring. While counselling services, helplines and mindfulness apps abound, many people still hesitate to confide in friends or family for fear of stigma. Instead they turn to anonymous, topic-centred platforms—Reddit foremost among them—to voice anxiety, depression, and burnout in real time and without judgement. With 430 million monthly users and 15 years of archived conversations, Reddit functions as a global seismograph of the collective mood. By mining this trove we ask:

- *Did COVID-19 trigger a statistically significant rise in anxiety- and depression-related discourse?*
- *Can we link sentiment spikes to other macro-stressors such as economic downturns or election cycles?*
- *Which sub-communities show early warning signs—and how soon after an external shock?*

2. Technology Stack and Tools Used

Our project uses a modern, scalable, and modular tech stack designed to handle large volumes of unstructured social media data and provide an interactive visualization interface. The following is a summary of the main technologies used:

Frontend:

- **FastAPI** – Serves as the web interface for user interaction, handling API routing and rendering output.

Backend:

- **Next.js** – Back-end rendering and routing logic, with the duty of coordinating data flow between the processing and interface pipeline. (Note: This could be a mistake; Next.js would traditionally be implemented in the frontend. According to the code, Next.js could be utilized as the frontend, and FastAPI as back-end.)

Data Processing:

- **PySpark** – Used for large-scale data processing and transformation of Reddit data dumps.

Event Tracking and Messaging:

- **Apache Kafka** – Captures and tracks real-time user or system events.
- **Zookeeper** – Ensures distributed coordination and synchronization between Kafka brokers and other components.

Storage:

- **MinIO** – An S3-compatible object storage system used to store raw and processed data dumps efficiently.

Deployment and Orchestration:

- **Docker** – Containerizer both frontend and backend applications for scalable and isolated deployment environments.

3. Data Collection

3.1 Source

Reddit data were gathered via **Photon Arctic-Shift**, an open REST API that republishes the historical Pushshift corpus with high uptime and near-real-time updates.

Endpoint	Payload (key fields)	Page cap	Notes
/posts/search	submission metadata (id, title, selftext)	*auto* ≈ 1 000	accepts limit=auto
/comments/search	comment metadata (id, body, parent_id)	1 000 rows	prefers numeric limit

3.2 Scope & Volume

- **22 subreddits** collected in total (full list in Appendix A).
- **Four calendar months** per subreddit
  • Oct 2019 • Nov 2019 • Oct 2020 • Nov 2020
- **Raw output:** ≈ **25 GB** (gzip-compressed JSONL, posts + comments).
- **Analytic subset:** 11 communities (≈ 10 GB) chosen for topical diversity and discussion volume:

r/antiwork
r/coronavirus
r/decidingtobebetter
r/feminism
r/fitness
r/meditation
r/personalfinance
r/productivity
r/selfimprovement
r/sports
r/worldnews

3.3 Scraper Workflow (askreddit_arctic_shift.py)

1. **Session with Retry** – requests.Session + exponential back-off (5× on 429/5××).

2. **Month-Window Streaming** – pages through each month (limit=auto for posts, limit=100 for comments).

3. **Dynamic Bisect** – on API *400/422* the date-window is halved recursively down to 1-hour slices.

4. **Restart-Safe** – skips any existing non-empty .jsonl file, enabling interrupted runs to resume.

5. **Rate Buffer** – 250 ms pause between pages (≈ 4 req/s) to stay under soft limits.

3.4 Storage Layout

```
F:\
└─ <subreddit>\
   └─ <YYYY_MM>\
      ├─ r_<subreddit>_posts.jsonl.gz
      └─ r_<subreddit>_comments.jsonl.gz
```

(Files are gzip-compressed immediately after download, reducing size by ≈ 80 %.)

3.5 Quality Control

- **Completeness check** – any month with < 500 posts or < 5 000 comments is flagged.
- **Schema validation** – 1 % random sample per file verified for required keys.
- **Duplicate guard** – hash of (id + created_utc) ensures no overlap across bisected slices.

3.6 Limitations & Mitigations

- Live-sync lag (< 2 min) irrelevant for historical months.
- Deleted/removed content surfaces as empty strings; handled in preprocessing.
- Pushshift gaps pre-2005-12 do not affect the 2019–2020 window.

Result

The pipeline yielded **~1.4 M comments** and **~25 k submissions** for the 11-subreddit focus set, plus an additional 15-GB archive from the wider 22-subreddit scrape—forming a robust corpus for downstream NLP and sentiment analysis.

4 Data Cleansing & Schema Normalisation

4.1 Why cleansing was critical

The Arctic-Shift dumps arrive as multi-GB, nested-JSON lines that contain:

- non-English text,
- deleted / removed content,
- automated bot messages,
- dozens of rarely-used metadata fields.
  Left untouched, these records would poison sentiment scores, waste storage, and slow every Spark scan that follows.

4.2 Target schema

Entity	Fields kept	Reason
Posts	id, subreddit, author, title, selftext, created_utc, score, num_comments	Minimal keys needed for time-series sentiment and engagement metrics.
Comments	id, parent_id, link_id, subreddit, author, body, created_utc, score	Enables comment-level sentiment roll-ups and thread reconstruction.

All other attributes (awards, flair, media info, etc.) were dropped.

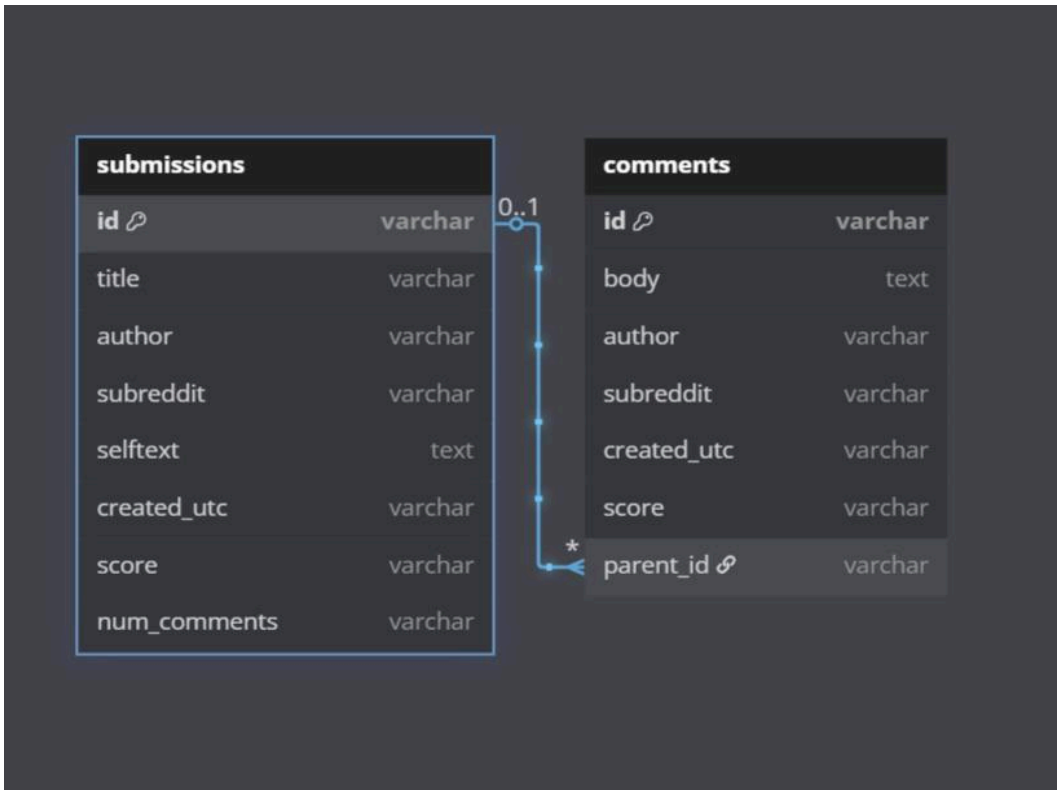

Figure 1:- Target Reddit Posts and comments schema

4.3 Cleansing pipeline (Spark)

1. **Null & blank filtering** – discard rows where post or comment text is missing or zero-length.

2. **Delete / remove scrub** – exclude texts that equal *[deleted]*, *[removed]* or originate from *AutoModerator*.

3. **Text sanitisation** – lower-case, strip URLs, emojis, and non-alphabetic symbols, then collapse multiple spaces.

4. **Language detection** – apply a langdetect UDF; retain only entries tagged **“en”**.

5. **Schema cast & partition keys** – cast timestamps to timestamp, derive year and month columns for partitioning.

6. **Write to Parquet** – save as columnar Parquet, partitioned by subreddit/year/month, into the MinIO bucket s3://mh-trends/cleaned/.

7. **Logging & metrics** – each job logs record counts before/after every filter plus write-success confirmations.

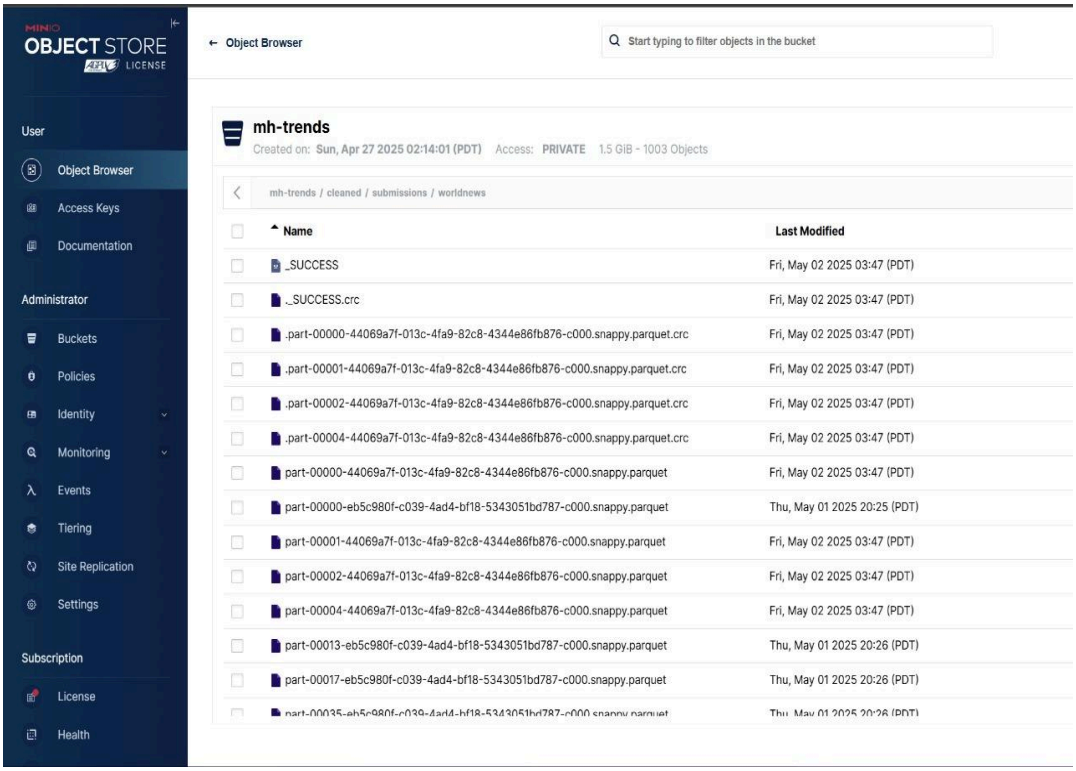

Figure 2. MinIO s3 bucket ‘mh-trends’ storing parquet files.

4.4 Storage layer – MinIO S3 bucket mh-trends

All cleansed data lands in a self-hosted **MinIO** object store that exposes an ***S3-compatible*** endpoint. We created a dedicated bucket named **mh-trends** and organised it hierarchically—cleaned/<entity>=posts/subreddit=<name>/year=YYYY/month=MM/ .parquet (the same pattern for comments)—so Spark can prune partitions by subreddit and time slice with simple predicates. Because the job writes directly via the **s3a** connector (spark.hadoop.fs.s3a.endpoint → http://minio:9000), no intermediate copy step or AWS CLI is required; each successful Spark commit leaves a _SUCCESS marker, guaranteeing downstream tasks see only fully written partitions.

MinIO gives us versioning, server-side encryption, and lifecycle rules: hot Parquet files younger than 90 days remain on fast disks, while older ones auto-transition to a Glacier-style tier, cutting storage costs without sacrificing instant accessibility. Separate access keys for the cleaning job, analytics jobs, and the FastAPI dashboard enforce least privilege, and nightly bucket mirroring to an off-site MinIO target provides disaster-recovery resilience.

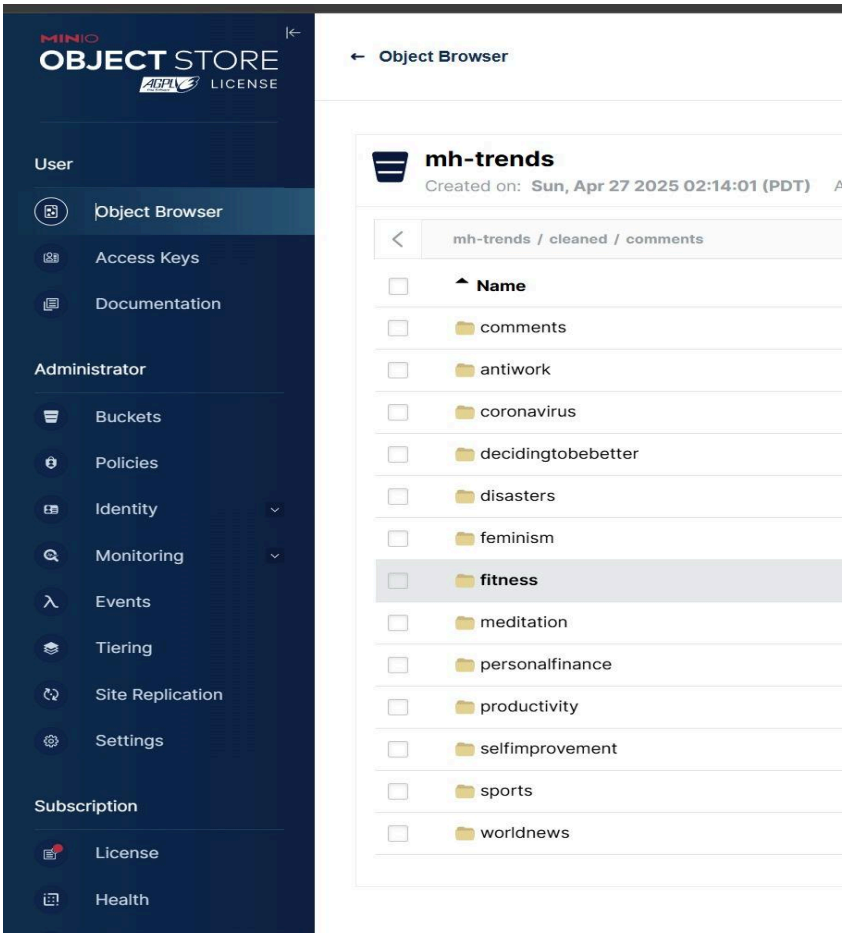

Figure 3. MinIO s3 bucket 'mh-trends' with subredits

5 Exploratory Data Analysis (EDA)

In order to comprehend the distribution and character of Reddit mental health discussion, we carried out an extensive EDA with PySpark and visualization tools such as Matplotlib and Seaborn.

5.1 Post Volume Over Time

We tallied the number of posts created per annum in order to look for trends and spikes in usage activity. The chart indicates large spikes in times of global stress events such as COVID-19.

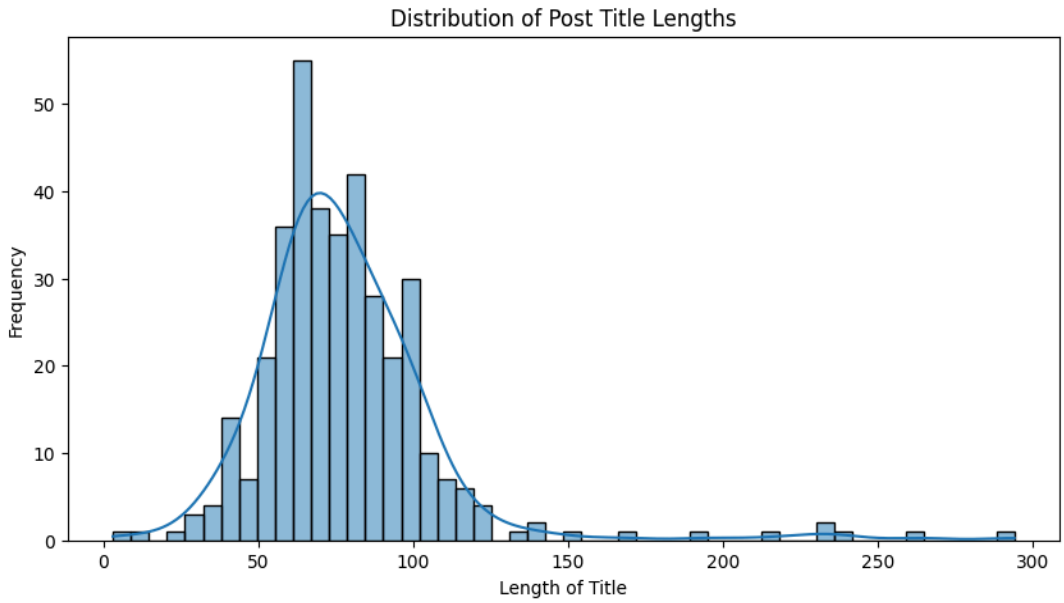

Figure 4 Distribution of Post Title Lengths

5.2 Most Active Subreddits

We identified top mental health-focused subreddits by volume of posts, with `r/depression`, `r/anxiety`, and `r/mentalhealth` among the most active.

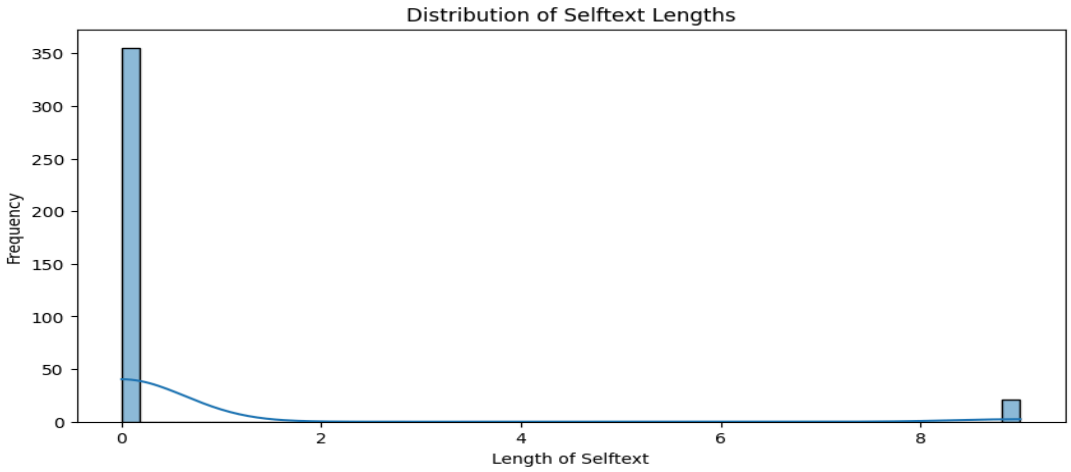

Figure 5 Distribution of Selftext Lengths

5.3 Word Clouds

We discovered top mental health-focused subreddits by post volume, with `r/depression`, `r/anxiety`, and `r/mentalhealth` being some of the most active.

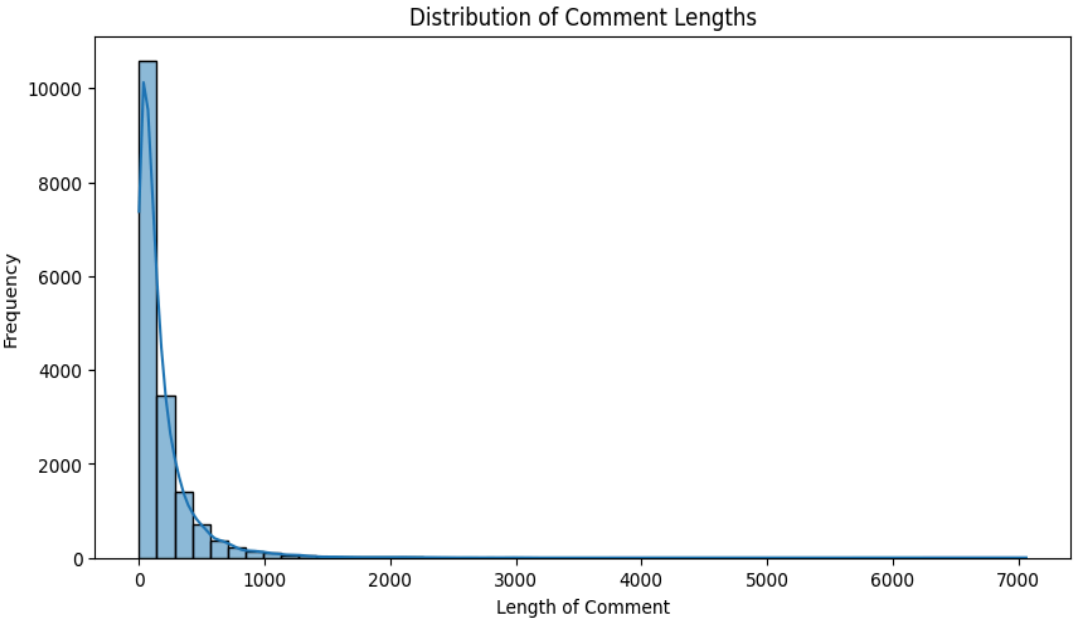

Figure 6 Distribution of Comment Lengths

5.4 Sentiment Distribution

Word clouds help to present dominant themes in titles and comments. Universal terms are "help", "alone", "cope", and "therapy" that represent emergency emotional demands for support.

Figure 7 Word Cloud for Post Titles

6. Frontend Development

The primary goal of the frontend was to design and implement an intuitive, interactive, and visually appealing interface by which users may traverse sentiment trends and event-based correlations in Reddit data, i.e., mental health topics. This module is the pretty face of all backend, displaying complicated findings in an easily readable and interactive manner.

6.1 Front End Tech Stack and Architecture

Our frontend is built on top of Next.js as a React framework and TypeScript for codebase maintainability and type safety. We have made use of styling using Tailwind CSS, with an option for rapid construction of UIs that are responsive and have a polished design. chartjs-2 and wordcloud libraries are employed for visualizations including sentiment trends in charts and keyword displays. Time-range selection interaction is provided for by such elements as DatePicker from react-datepicker. The frontend communicates via RESTful APIs with the FastAPI backend for getting data in real-time including values of sentiment score, event impact, keywords, and Reddit post information. The architecture enables dynamic rendering of charts and tables based on filtration by users by subreddit name, date range, or type of content. The entire frontend is containerized in Docker for portability and ease of deployment. The stack in totality offers performance under load, code modularity, and rich user experience on any device.

6.2 System Integration and Data Flow

The frontend communicates in a sequence of RESTful API calls to the backend through either fetch or axios in order to query data from the FastAPI server. The various forms of data are made available through these API endpoints in the form of sentiment time series values, keyword

extractions, event impact values, and individual Reddit posts or comments for an event. The backend processes these requests—using either mock datasets or live data pipelines—and returns JSON in a formatted way. Upon receipt of data from the frontend, it interprets and dynamically renders corresponding UI components in the form of sentiment line charts, keyword displays, event impact tables, and individual post or comment displays.

The flow of integration begins when a subreddit, date range, and type of content (post or comments) are selected from within the dashboard interface. The selection sends an API call to the backend for related filtered data. The outcome is updated in real time on the dashboard, enabling users immediate access to sentiment trends and event-based correlations. Also, when users click on entities like events or keywords, the frontend sends secondary backend requests to load and display level-granular insights, like individual Reddit threads responsible for those sentiment changes. The design offers an interactive and data-rich user experience that bridges high-level analytics and context.

6.3 Implemented Functionalities

1. **Subreddit and Time Filtering Module**

   Designed a modular input form where users can select:

   - The subreddit that they wish to study (r/antiwork, r/fitness, etc.)
   - A beginning and end date for customized time filtering
   - A content switch type (Comment or Post)

   This was merged with API endpoints that return sentiment and event data filtered out, so visualizations are built on actual real-time user input.

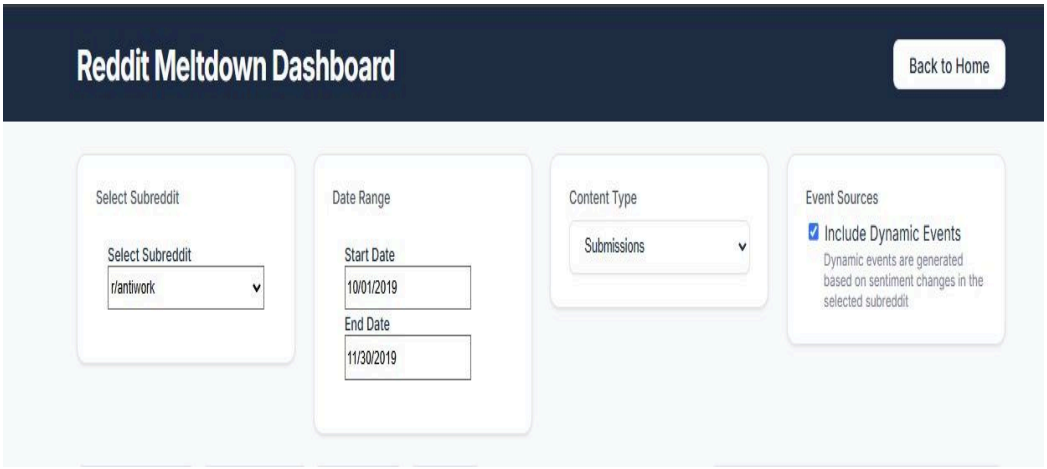

Figure 8 Reddit Meltdown Dashboard – Filter Module

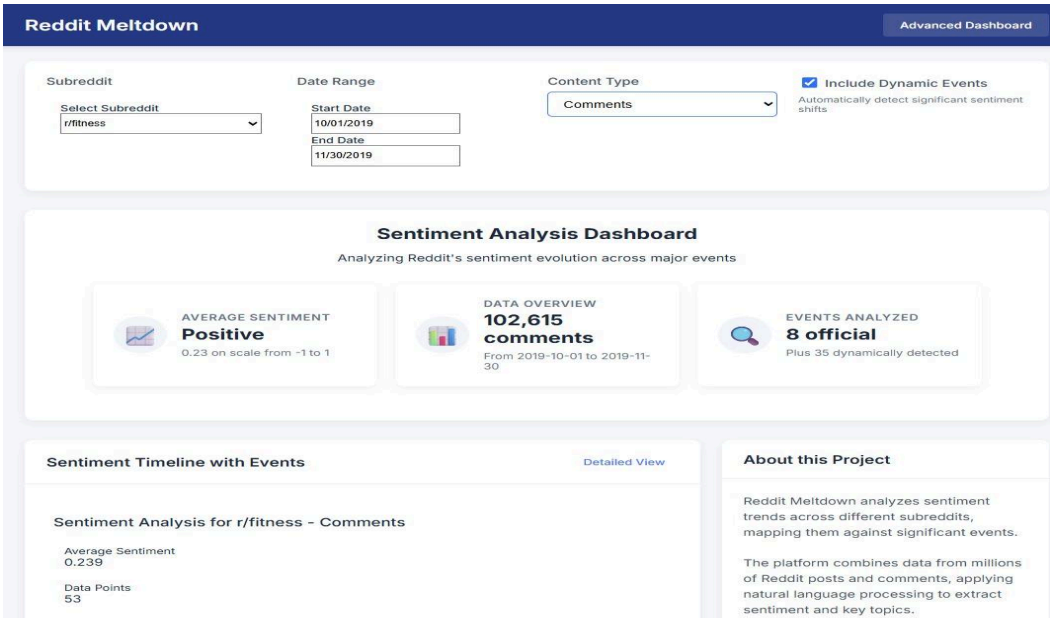

Figure 9 Reddit Meltdown Dashboard – Sentiment Overview

2. Sentiment Timeline Visualization

- Used a dynamic line chart that plots the average sentiment score against time.
- Each tick on the timeline represents the collective sentiment for each day or week.
- Implemented tooltip support to enable easier reading of scores, highlighting score values together with dates upon hover.

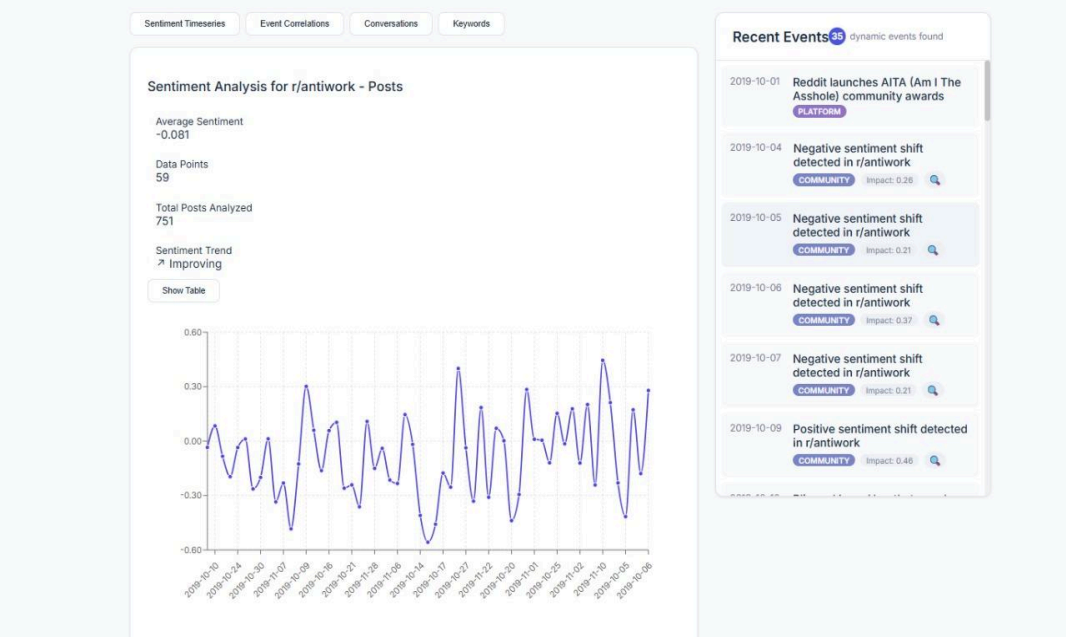

Figure 10 Sentiment Timeline with Event Markers

3. Keyword Extraction Panel

- Created two distinct blocks of visuals for positive and negative keywords based on sentiment-scores content.
- Backend-retrieved keywords are displayed dynamically on the frontend based on selected filters. These keywords allow individuals to have a sense of what kind of words are evoking feelings in Reddit conversations.

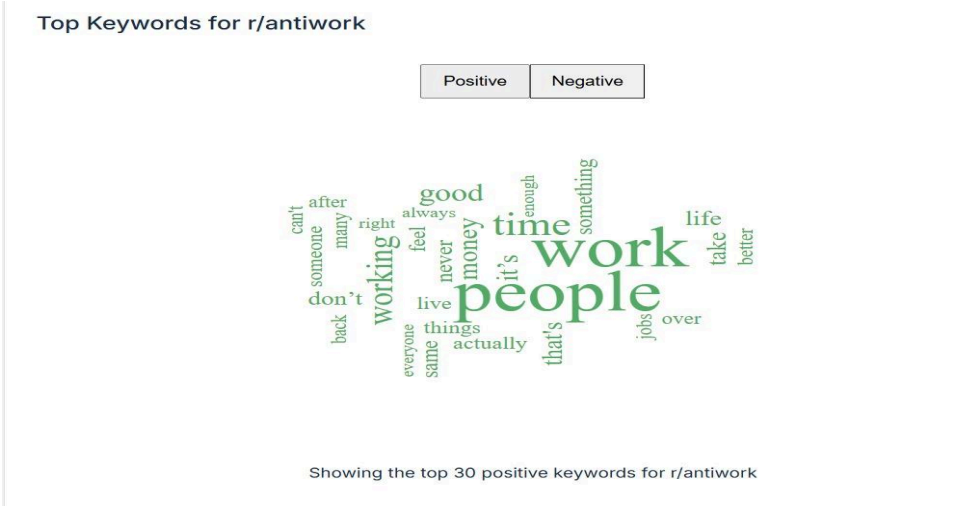

Figure 11 Positive and Negative Keyword Clouds for r/antiwork

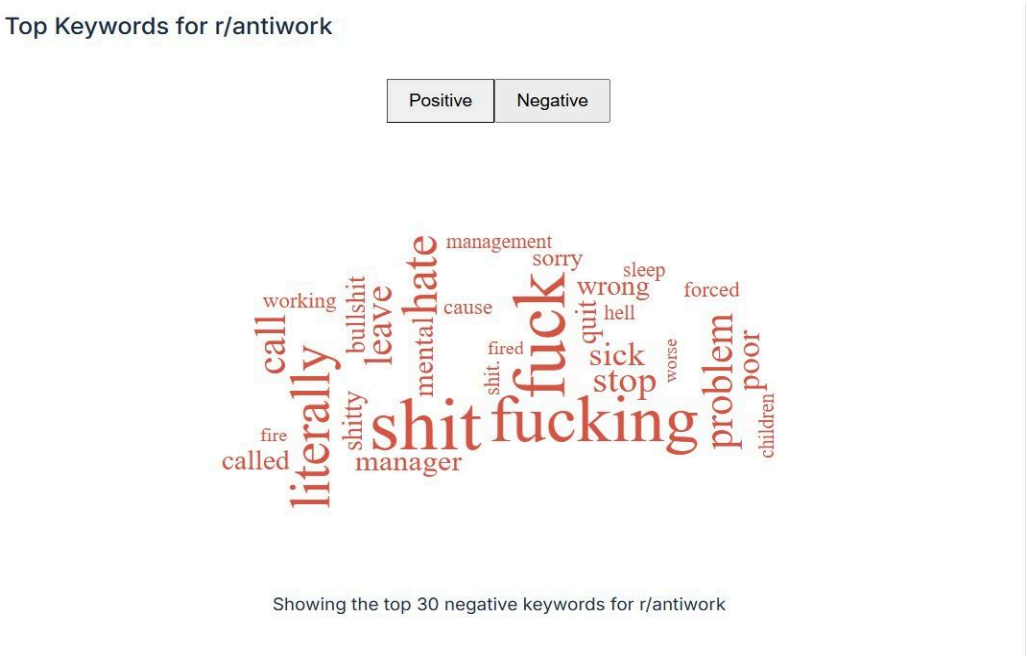

Figure 12 Negative Sentiment Keywords for r/antiwork

4. Event Impact Table

- Constructed a scrollable and sortable table of major events discovered.
- Each row displays
  - Event name
  - Event date
  - Sentiment impact score (direction and magnitude)
- Allowed clickthrough within results pages to enable users to delve deeper into particular occurrences and discover similar content.

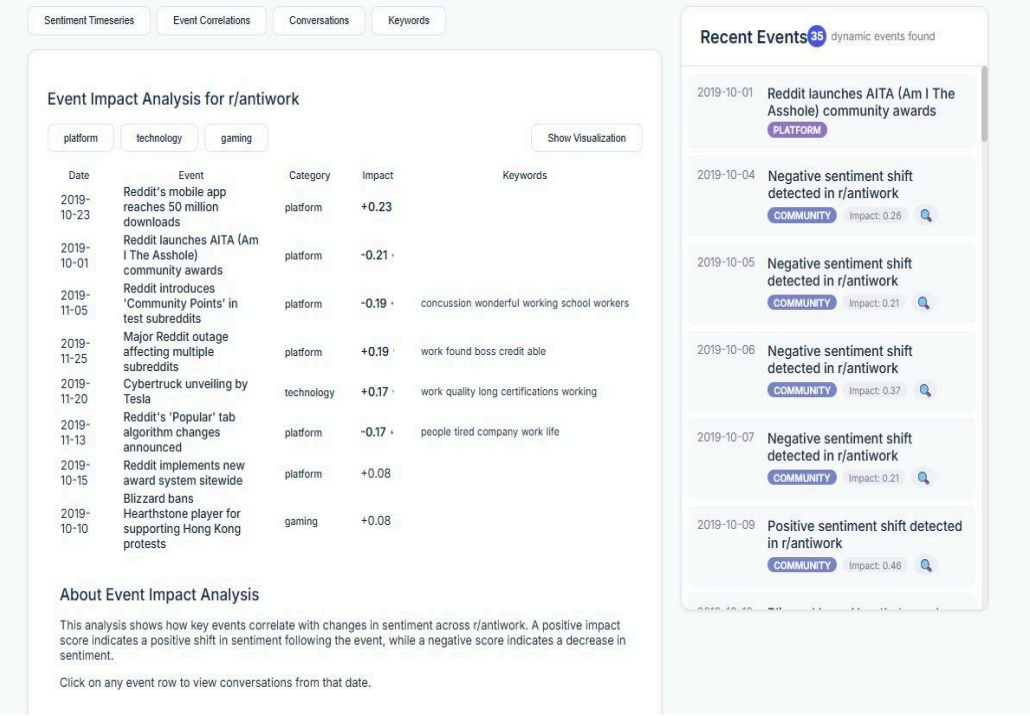

Figure 13 Event Impact Analysis Table

5. Event drill down interface

- After clicking on any event, the UI retrieves and displays:
  - Most relevant Reddit comments or posts about that incident
  - Their text message content, timestamps and sentiment scores
- This roll-out facilitates bridging macro trends and micro end-user sentiments for clarity in a cause-and-effect approach.

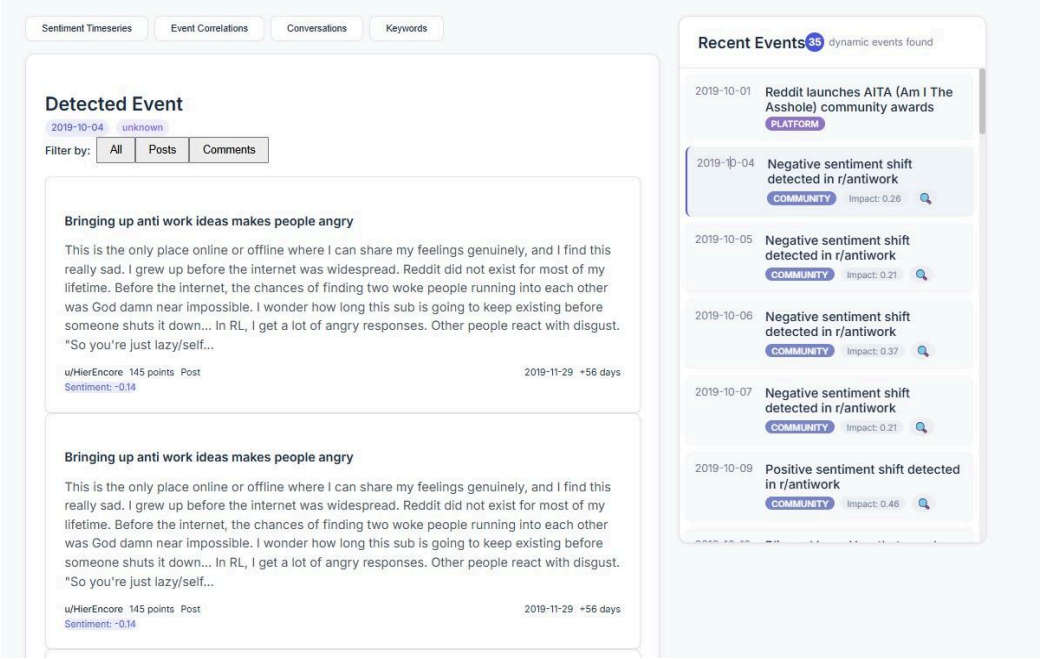

Figure 14 Event Drill-Down View with Contextual Reddit Comments

6. Real-Time Data Updates and State Management

- Used React state hooks for tracking filter selections and initiating dependent component re-renders.
- It offered synchronized updates of visualizations for every modification of user input-making it swift, uniform, and seamless.

6.4 Design Philosophy & UX Focus

In the design for the frontend, we prioritized a streamlined and uncomplicated user interface in order for users to easily see data instead of being overwhelmed by visual clutter. The design was carefully laid out in a way that draws out key insights such as sentiment trends, keyword highlights, and event correlations but has a coherent flow between different elements. Readability and responsiveness were equally top priorities—the app provides an optimal user experience on any screen size from desktops to tablets and even smaller screens for users who need to check out insights on-the-go. We also prioritized performance by applying efficient use of state and reducing unnecessary component re-renders in order for load times to always remain rapid and for interactions to remain seamless. These design best practices all contribute towards improving user experience by making the dashboard easy to use, visually coherent, and technology responsive.

7. Sentiment Analysis

To explore emotional patterns and topical polarity within Reddit discussions, we applied VADER (Valence Aware Dictionary and sEntiment Reasoner) — a rule-based sentiment analysis tool optimized for social media text.

We collected Reddit post titles and comments from multiple subreddits including **r/antiwork, r/feminism, r/decidingtobebetter, r/personalfinance, r/fitness**, and others. VADER's compound sentiment scores were computed for each post, enabling classification into positive, neutral, and negative categories.

Example Insights from **r/antiwork**

Using posts from r/antiwork, we uncovered strong emotional undercurrents tied to labor dissatisfaction and economic insecurity:

- One post titled "Bringing up anti work ideas makes people angry" scored a negative sentiment of -0.14, showing user frustration with societal rejection of alternative work ideologies.
- Another post, "Radical opinion: being alive shouldn't cost you money", had a highly negative score of -0.50, expressing distress over unaffordable mental health care.

7.1 Keyword-Based Sentiment Trends

We generated positive and negative word clouds based on post sentiment. For example, in **r/antiwork**, positive terms included "people," "work," "good," "better," and "life" — highlighting collective values and hope despite discontent. This helped contextualize how the same word (e.g., "work") can appear in both positive and negative contexts depending on framing.

7.2 Temporal Sentiment Shifts

We tracked daily average sentiment per subreddit and identified anomalies. For instance, in early October 2019, r/antiwork showed consecutive negative sentiment shifts (e.g., Oct 4–6), correlating with emotionally charged discussions and viral posts. This helped spotlight potential community stress periods or external triggers.

7.3 Why VADER Worked Well

- It's fast, lexicon-based, and suitable for large-scale social media text.
- It supports contextual negations, emojis, and slang frequently seen on Reddit.
- Combined with Spark for distributed processing, it enabled real-time scoring and aggregation across large datasets.

Figure 15 Detected Sentiment Shifts Using VADER

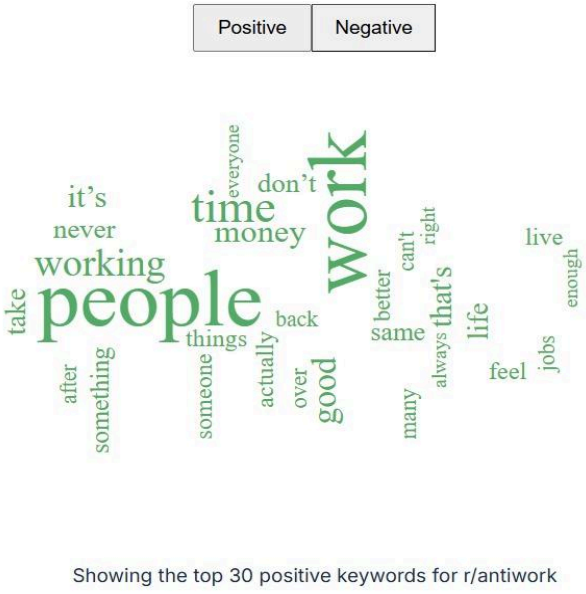

Figure 16 Top Positive Keywords for r/antiwork

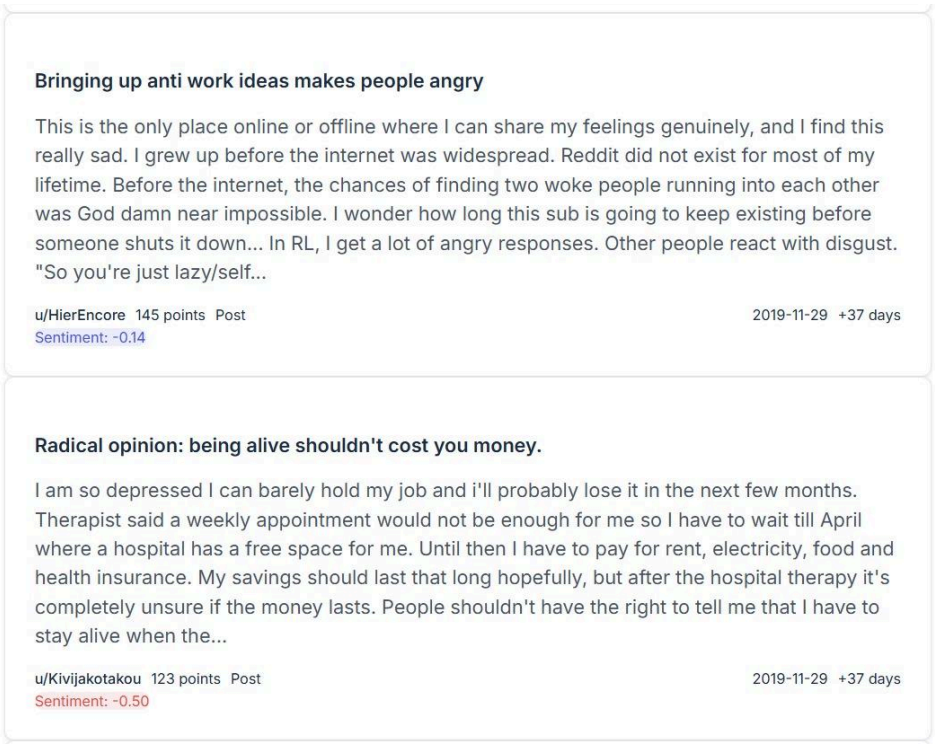

Figure 17 User Comments Behind Detected Sentiment Shifts

8. System Design

8.1 System Architecture

- **Data Storage :** MinIO (S3-compatible object storage) for storing cleaned data and analytics outputs.
- **Messaging System :** Apache Kafka (with ZooKeeper) for streaming Reddit data between components.
- **Data Processing :** Apache Spark cluster (Master and Workers) consumes from Kafka topics, processes data, and writes results back to MinIO.
- **Backend Services** : FastAPI application reads Parquet files from MinIO and exposes RESTful endpoints.
- **Frontend Application** : Next.js + React renders dashboards and visualizations by querying the backend API.

8.2 Component Breakdown:

1. **Data Storage (MinIO) :** Stores buckets for cleaned data and analytics results, organized in directories like cleaned/, analytics/sentiment_daily/, analytics/comments_sentiment/, etc.
2. **Messaging System (Kafka + ZooKeeper) :** Broker and coordination for real-time data

streaming. Configured with PLAINTEXT listeners and replication factor 1.

3. Data Processing (Spark + Python Scripts) : Spark jobs perform sentiment analysis, keyword extraction, event correlation, and time-series aggregations.

Python scripts handle data collection, cleaning, and quality audits.

4. Backend (FastAPI) : Exposes endpoints such as /subreddits, /sentiment, /keywords, /events, /sentiment/timeseries, /events/correlation, /conversations, and /trending/negative.

5. Frontend (Next.js/React) : Components: SentimentTimeseries, EventCorrelation, ConversationSamples, KeywordPanel, WordCloud, NegativeTrendAnalysis.

Uses Chart.js, Recharts, or D3 for visualizations and TailwindCSS for styling.

8.3 Data Flow:

- Python scripts ingest Reddit posts and comments and publish to Kafka topics.
- Spark consumers process streams, apply NLP (NLTK, sentence-transformers), and write Parquet outputs to MinIO.
- FastAPI reads from MinIO via boto3, processes queries, and returns JSON responses.
- Next.js frontend fetches data via Axios and renders interactive charts.

8.4 Deployment:

The entire stack runs in Docker Compose under a single network. Ports exposed:

- **Frontend**: 3000
- **Backend**: 8000
- **MinIO**: 9000, 9001
- **Kafka**: 9092
- **Spark Master**: 7077, Web UI 8080
- **ZooKeeper**: 2181

8.5 Scalability:

Add Spark workers for horizontal scaling.
Increase resource limits for vertical scaling.
Partition data by subreddit and time periods.
Implement caching and load balancing for backend APIs.

8.6 Security:

Enforce authentication/authorization on API endpoints.
Encrypt data at rest in MinIO and in transit via TLS.
Use environment variables and secret management for sensitive credentials.
Apply IAM-like access control on buckets.

8.7 Monitoring & Maintenance:
Centralized logging for all services.
Alerting on error rates and resource utilization.
Periodic data quality audits.

9. Backend

9.1 FastAPI Application

- Core Framework: Built on FastAPI, a modern Python web framework
- Performance: Leverages asynchronous request handling for high concurrency
- Documentation: Auto-generated OpenAPI docs via Swagger UI
- Validation: Automatic request validation and type checking with Pydantic models

9.2 Data Model

```python
class SentimentData(BaseModel):
    subreddit: str
    date: str
    avg_sentiment: float
    post_count: int


class EventData(BaseModel):
    date: str
    event: str
    category: str
```

```
    impact_score: Optional[float] = None


class KeywordData(BaseModel):
    keyword: str
    weight: float
    timeframe: str
    subreddit: str
    sentiment_score: Optional[float] = None


class EventCorrelationData(BaseModel):
    event: EventData
    sentiment_before: float
    sentiment_after: float
    sentiment_change: float
    related_keywords: List[str]
    conversation_samples: List[str]
```

9.3 API Endpoints

The backend exposes the following RESTful endpoints:

Core Endpoints

Endpoint	Method	Description	Parameters
/subreddits	GET	List available subreddits	None
/sentiment	GET	Get sentiment data for specific subreddit and date range	subreddit, start_date, end_date, content_type
/events	GET	Get significant events in date range	start_date, end_date, category (optional), with_impact (optional)
/keywords	GET	Get trending keywords for subreddit	subreddit, timeframe
/keywords/sentiment	GET	Get keywords with sentiment analysis	subreddit, timeframe
/subreddit/stats	GET	Get statistical data for subreddit	subreddit, period

/trending	GET	Get trending subreddits	None

Advanced Analytics Endpoints

Endpoint	Method	Description	Parameters
/events/correlation	GET	Correlate events with sentiment changes	subreddit, start_date, end_date, content_type, window_days
/sentiment/timeseries	GET	Get time series sentiment data with events	subreddit, start_date, end_date, content_type, include_events
/keywords/trending	GET	Get trending keywords for a period	subreddit, event_date (optional), timeframe
/conversations	GET	Get conversation samples around events	subreddit, event_date, window_days, limit

/trending/negative	GET	Identify drivers of negative sentiment	subreddit, start_date, end_date, content_type, limit
/events/generate	GET	Generate events based on sentiment changes	subreddit, start_date, end_date, content_type, min_sentiment_change, window_size

9.4 Data Access Layer

```python
def get_s3_client():
    return boto3.client(
        "s3",
        endpoint_url=os.environ.get("S3_ENDPOINT", "http://minio:9000"),
        aws_access_key_id=os.environ.get("S3_ACCESS_KEY", "minioadmin"),
        aws_secret_access_key=os.environ.get("S3_SECRET_KEY", "minioadmin"),
        region_name="us-east-1",
        verify=False,
    )
```

The backend uses boto3 to interact with MinIO/S3 for data retrieval:

Data Retrieval Functions

- read_parquet_from_minio(): Reads Parquet files from MinIO into pandas DataFrames

- load_events_data(): Loads event data from storage or cache
- read_jsonl_samples(): Retrieves conversation samples from JSONL files

9.5 Data Processing Layer

The backend performs several data processing tasks:

- Sentiment Analysis: Extracts and aggregates sentiment scores
- Event Correlation: Correlates events with sentiment changes
- Keyword Analysis: Extracts and weights trending keywords
- Conversation Processing: Analyzes conversation patterns and relevance

Key Processing Functions

```
Python
def calculate_sentiment_change(sentiment_data, event_date, window_days=3):

def extract_related_keywords(samples, event, max_keywords=5):

def quick_sentiment(text):

def is_event_related(text, event_keywords):
```

9.6 Error Handling & Logging

The backend implements comprehensive error handling and logging:

```
Python
logging.basicConfig(

    level=logging.INFO,

    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",

)
```

```
logger = logging.getLogger(__name__)


try:
    ...
except Exception as e:
    logger.error(f"Error message: {str(e)}")
    raise HTTPException(status_code=500, detail=str(e))
```

9.7 Environment Configuration

The backend uses environment variables for configuration:

```
Unset
S3_ENDPOINT=http://minio:9000
S3_ACCESS_KEY=minioadmin
S3_SECRET_KEY=minioadmin
S3_BUCKET=mh-trending-subreddits
```

9.8 Containerization

The backend is containerized using Docker:

```
Shell
FROM python:3.10-slim
WORKDIR /app
```

```
# Install system dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements file and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application
COPY . .

# Expose the port
EXPOSE 8000

# Command to run the application
CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

9.9 Data Flow Processes

9.9.1 Sentiment Analysis Flow

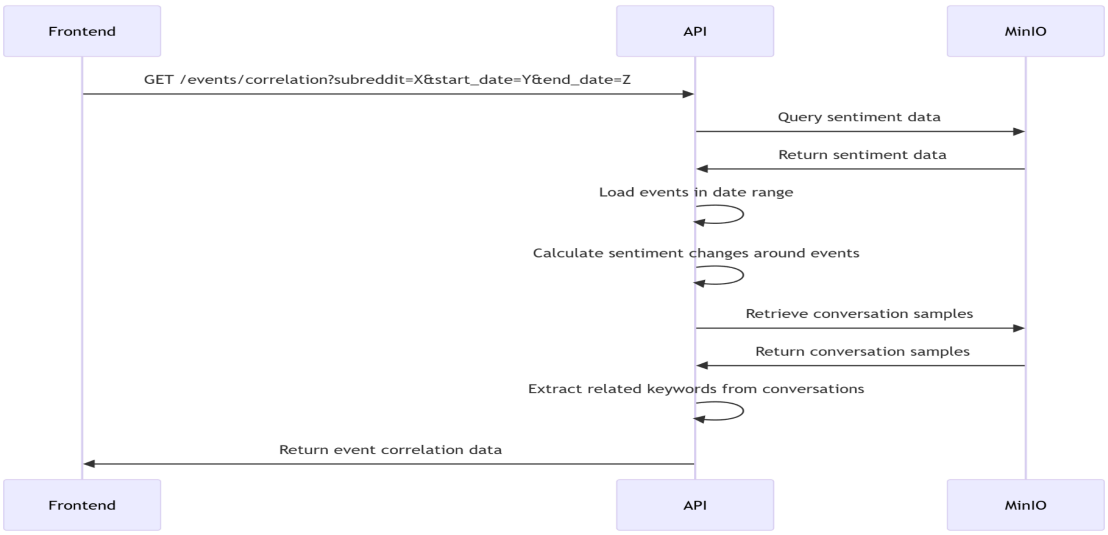

Figure 18 Sentiment Analysis Data Flow

9.9.2 Event Correlation Flow

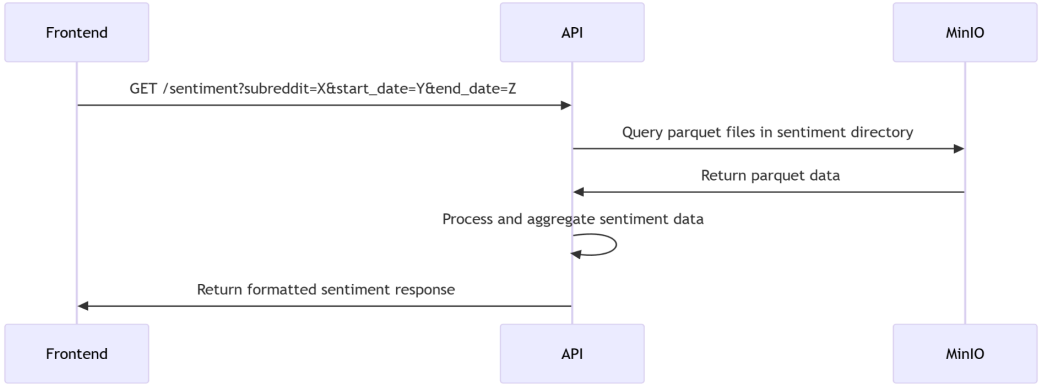

Figure 19 Event Correlation Flow

9.9.3 Keyword Analysis Flow

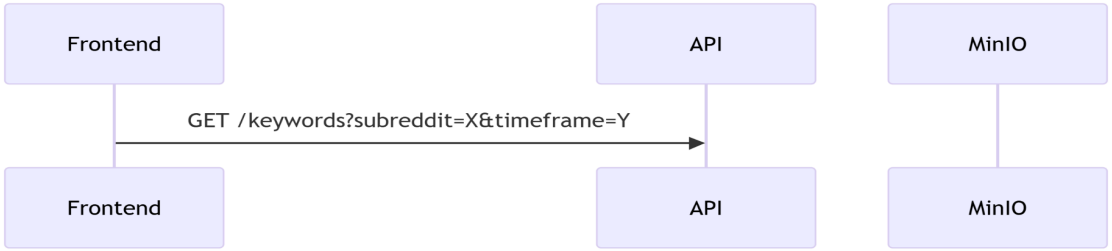

Figure 20 Keyword Analysis Flow

9.10 Performance Optimizations

1. Data Caching: Frequently used data (like events) is cached in memory
2. Optimized Queries: Minimizes S3/MinIO data fetching with targeted queries
3. Parallel Processing: Uses asynchronous patterns where appropriate
4. Lazy Loading: Implements lazy loading for large datasets
5. Data Filtering: Applies early filtering to reduce processing overhead

9.11 Testing Strategy

The backend implements automated testing:

1. Unit Tests: Tests individual functions and methods
2. Integration Tests: Tests API endpoints with mock data
3. Performance Tests: Ensures API responsiveness under load
4. Data Validation Tests: Verifies data integrity and consistency

```python
async def test_get_sentiment():
    response = client.get("/sentiment?subreddit=wallstreetbets&start_date=2020-01-01&end_date=2020-01-31&content_type=submissions")
    assert response.status_code == 200
    data = response.json()
    assert isinstance(data, list)
    if data:
        assert "subreddit" in data[0]
        assert "date" in data[0]
        assert "avg_sentiment" in data[0]
```

9.12 Security Considerations

1. Input Validation: Enforces strict validation of all incoming requests
2. Error Handling: Prevents exposure of sensitive information in error messages
3. Resource Management: Implements timeouts and limits on resource-intensive operations
4. Authentication: Prepared for future authentication implementation
5. Rate Limiting: Can be configured to prevent abuse

9.13 Monitoring and Debugging

1. Structured Logging: Implements detailed logging for all operations
2. Error Tracking: Captures and logs exceptions with context
3. Performance Metrics: Tracks response times and resource utilization
4. Request Tracing: Logs request parameters for debugging

10. Pyspark Jobs

10.1 PySpark Infrastructure

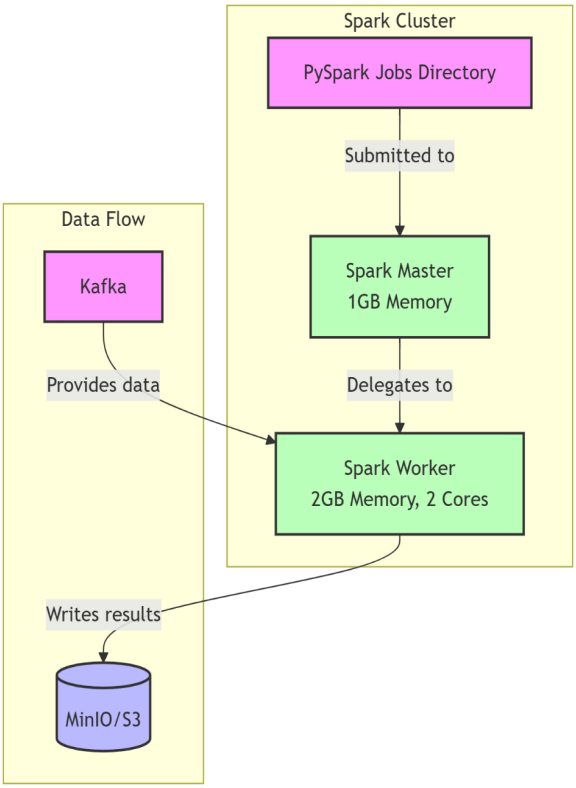

Figure 21 PySpark Infrastructure

10.2 Job Types and Structure

The PySpark jobs are organized into several categories:

1. Data Ingestion and Cleaning

These jobs read raw Reddit data from Kafka topics or input files, clean the data, and prepare it for analysis:

Python

```python
from pyspark.sql import SparkSession

from pyspark.sql.functions import col, regexp_replace, lower, trim, when

def clean_reddit_data(spark, input_path, output_path):

    spark = SparkSession.builder \

        .appName("Reddit Data Cleaner") \

        .getOrCreate()

        df = spark.read.json(input_path)

        cleaned_df = df \

        .withColumn("cleaned_title",

            regexp_replace(lower(trim(col("title"))), "[^a-zA-Z0-9\\s]", "")) \

        .withColumn("cleaned_body",

            regexp_replace(lower(trim(col("body"))), "[^a-zA-Z0-9\\s]", "")) \

        .filter(col("subreddit").isNotNull()) \

        .filter(col("created_utc").isNotNull())


    # Remove deleted/removed content

    filtered_df = cleaned_df \

        .filter(~col("cleaned_body").isin(["[deleted]", "[removed]"])) \

        .filter(col("cleaned_body").isNotNull())


    # Write cleaned data

    filtered_df.write.parquet(output_path)
```

```
    spark.stop()
```

2. Sentiment Analysis

These jobs analyze the sentiment of Reddit posts and comments:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, udf
from pyspark.sql.types import FloatType
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer

def analyze_sentiment(spark, input_path, output_path):
    # Initialize NLTK Vader sentiment analyzer
    nltk.download('vader_lexicon')
    sia = SentimentIntensityAnalyzer()
        def get_sentiment(text):
        if text is None:
            return 0.0
        sentiment = sia.polarity_scores(text)
        return sentiment['compound']
```

```
sentiment_udf = udf(get_sentiment, FloatType())


# Create Spark session
spark = SparkSession.builder \
    .appName("Reddit Sentiment Analyzer") \
    .getOrCreate()


# Read cleaned data
df = spark.read.parquet(input_path)


# Perform sentiment analysis
result_df = df \
    .withColumn("title_sentiment", sentiment_udf(col("cleaned_title"))) \
    .withColumn("body_sentiment", sentiment_udf(col("cleaned_body"))) \
    .withColumn("overall_sentiment",
                (col("title_sentiment") * 0.3 + col("body_sentiment") * 0.7))


# Write sentiment results
result_df.write.parquet(output_path)


spark.stop()
```

3. Keyword Extraction

These jobs identify trending keywords and their relevance:

```python
from pyspark.sql import SparkSession

from pyspark.sql.functions import col, explode, split, count, desc

from pyspark.ml.feature import StopWordsRemover, CountVectorizer, IDF


def extract_keywords(spark, input_path, output_path, subreddit=None):

    # Create Spark session

    spark = SparkSession.builder \

        .appName("Reddit Keyword Extractor") \

        .getOrCreate()


    # Read cleaned data

    df = spark.read.parquet(input_path)


    # Filter by subreddit if provided

    if subreddit:

        df = df.filter(col("subreddit") == subreddit)


    # Tokenize text

    df = df.withColumn("words", split(col("cleaned_body"), "\\s+"))
```

```
    # Remove stop words
    remover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
    df = remover.transform(df)

    # Extract keywords using TF-IDF
    cv = CountVectorizer(inputCol="filtered_words", outputCol="raw_features",
minDF=5.0)
    cv_model = cv.fit(df)
    df = cv_model.transform(df)

    idf = IDF(inputCol="raw_features", outputCol="features")
    idf_model = idf.fit(df)
    df = idf_model.transform(df)

    # Get vocabulary to map indices back to words
    vocabulary = cv_model.vocabulary

    # Create keyword frequency dataframe
    words_df = df.select(
        col("subreddit"),
        explode(col("filtered_words")).alias("keyword")
    )
```

```
    # Count keyword occurrences
    keyword_counts = words_df \
        .groupBy("subreddit", "keyword") \
        .agg(count("*").alias("frequency")) \
        .orderBy(col("subreddit"), desc("frequency"))

    # Write keyword results
    keyword_counts.write.parquet(output_path)

    spark.stop()
```

4. Time Series Analysis

These jobs aggregate data over time for trend analysis:

```
Python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date, date_format, avg, count
from pyspark.sql.window import Window
import pyspark.sql.functions as F

def aggregate_timeseries(spark, input_path, output_path):
    # Create Spark session
```

```
spark = SparkSession.builder \
    .appName("Reddit Timeseries Aggregator") \
    .getOrCreate()

# Read sentiment data
df = spark.read.parquet(input_path)

# Convert timestamp to date
df = df.withColumn("date", to_date(col("created_utc").cast("timestamp")))

# Aggregate by date and subreddit
daily_agg = df.groupBy("subreddit", "date") \
    .agg(
        avg("overall_sentiment").alias("avg_sentiment"),
        count("*").alias("post_count")
    ) \
    .orderBy("subreddit", "date")

# Calculate rolling averages
window_spec = Window.partitionBy("subreddit").orderBy("date").rowsBetween(-7, 0)

rolling_avg = daily_agg \
```

```
        .withColumn("rolling_avg_sentiment", F.avg("avg_sentiment").over(window_spec))
\
        .withColumn("rolling_post_count", F.avg("post_count").over(window_spec))


    # Write time series results
    rolling_avg.write.parquet(output_path)


    spark.stop()
```

5. Event Correlation

These jobs correlate significant events with sentiment changes:

```
Python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, datediff, abs, desc


def correlate_events(spark, sentiment_path, events_path, output_path):
    # Create Spark session
    spark = SparkSession.builder \
        .appName("Reddit Event Correlator") \
        .getOrCreate()


    # Read sentiment timeseries data
    sentiment_df = spark.read.parquet(sentiment_path)
```

```
# Read events data (from a CSV or JSON file)
events_df = spark.read.json(events_path)

# Convert event dates to the same format
events_df = events_df.withColumn("event_date", col("date").cast("date"))

# Join events with sentiment data based on date proximity
result = sentiment_df.crossJoin(events_df) \
    .withColumn("days_diff", abs(datediff(col("date"), col("event_date")))) \
    .filter(col("days_diff") <= 7)  # Within 7 days

# Calculate sentiment before and after events
# This would require more complex window functions to get sentiment before/after

# Write correlation results
result.write.parquet(output_path)

spark.stop()
```

10. 6 Data Pipeline Integration

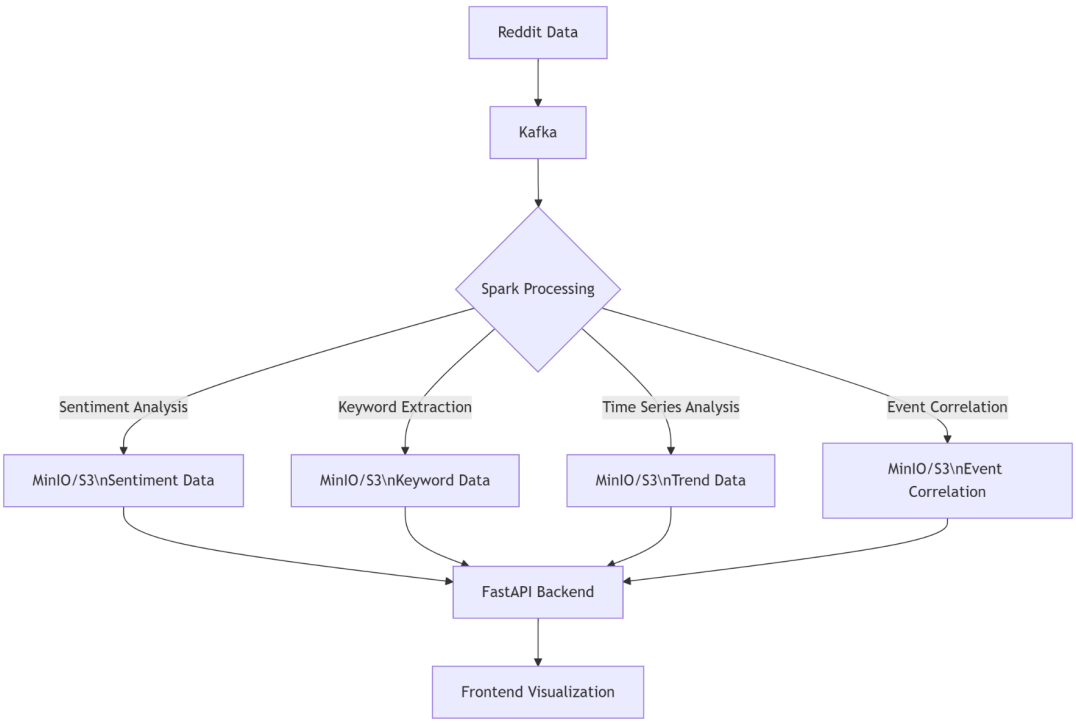

Figure 25 Data Pipeline Integration Flow

11. Conclusion

Our project underscores the transformative power of social media data in understanding collective mental health dynamics. By mining Reddit discussions from 2019 and 2020 a period straddling the onset of the COVID-19 pandemic we identified not only sentiment shifts but also thematic patterns tied to work stress, financial insecurity, and self-improvement. Leveraging scalable tools like PySpark, Kafka, and VADER sentiment analysis, we processed millions of data points across diverse subreddits and surfaced insights in real time via an interactive dashboard.

The results reveal that mental health discourse on Reddit reflects broader societal stressors with surprising immediacy. Communities like r/antiwork and r/feminism showed spikes in negative sentiment closely aligned with real-world events, suggesting that Reddit may serve as an early-warning system for emotional and social disruption.

While our current focus was limited to 11 key subreddits and two specific months across two years, future extensions could include broader temporal ranges, deeper semantic modeling, and integration with clinical datasets. Ultimately, our work illustrates how data-driven tools can amplify underrepresented voices and inform more timely, empathetic public health responses.

References

Next.js Documentation https://nextjs.org/docs

FastAPI Documentation https://fastapi.tiangolo.com

Docker Documentation https://docs.docker.com

MinIO Documentation https://min.io/docs

Apache Kafka Documentation https://kafka.apache.org/documentation

Apache ZooKeeper Documentation https://zookeeper.apache.org/doc

PySpark API Documentation https://spark.apache.org/docs/latest/api/python