INSIGHTS PLATFORM API

Twitter Conversation Analysis with Grok AI

Project Portfolio Document

FastAPI • Grok AI • SQLAlchemy • Docker

2025

Table of Contents

01 Project Overview

Insights Platform API is a production-ready backend system designed for real-time analysis of Twitter conversations using xAI's Grok AI model. The platform receives raw conversation data through a RESTful API, processes it asynchronously using AI-powered sentiment analysis and topic clustering, and provides structured insights through queryable endpoints.

Project Objectives

- Build a high-performance async API capable of handling 100+ requests/second
- Integrate Grok AI for intelligent sentiment analysis and topic categorization
- Implement robust rate limiting to manage both inbound traffic and external API calls
- Design an efficient background batch processing system for scalable data analysis
- Create a containerized deployment with strict resource constraints (2 CPU, 1GB RAM)
- Provide filtered and paginated insight retrieval with confidence-based scoring

Use Case Scenario

A customer service platform collects thousands of Twitter conversations daily. The Insights Platform API ingests this data, analyzes each conversation for sentiment polarity (-1.0 to +1.0), identifies topic clusters (e.g., "delivery_problems", "product_issues", "praise"), and assigns a confidence score. Support teams can then query insights filtered by time range, sentiment type, and confidence threshold to prioritize responses and identify emerging trends.

Metric	Value
API Framework	FastAPI (Python 3.11+)
AI Engine	Grok AI (xAI)
Database	SQLite (async via aiosqlite)
Inbound Rate Limit	100 requests/second
Grok API Rate Limit	10 calls/second
Batch Size	10 conversations per cycle
Max Query Results	1,000 per request
Deployment	Docker (2 CPU, 1GB RAM)

02 Key Features

RESTful API Design

Clean, well-documented REST API built with FastAPI. Supports conversation submission (POST) and insight retrieval (GET) with comprehensive request validation using Pydantic models. Follows HTTP semantics with proper status codes (202 Accepted, 429 Too Many Requests, 400 Bad Request).

Grok AI Integration

Deep integration with xAI's Grok model for advanced NLP analysis. Each conversation is analyzed for sentiment polarity (score from -1.0 to 1.0), topic clustering (identifying categories like "product_issues", "delivery_problems", "praise"), and confidence scoring. Structured prompt engineering ensures consistent JSON output from the AI model.

Asynchronous Batch Processing

Background batch processor that polls for pending conversations every 5 seconds, processes them in batches of 10 with parallel execution, and manages status transitions (pending -> processing -> completed/failed). Individual failures don't affect the batch, ensuring resilience.

Dual Rate Limiting

Two-tier rate limiting system: inbound API traffic is limited to 100 requests/second using a token bucket algorithm with sliding window, while outbound Grok API calls are throttled to 10 calls/second using semaphore-based concurrency control with time-window tracking.

Advanced Query Filtering

Insight retrieval supports powerful filtering: time range queries (ISO8601), sentiment type filtering (positive/negative/neutral), minimum confidence thresholds (0.0-1.0), and pagination with configurable limits (up to 1,000 results). Results are ordered by timestamp descending.

Docker Containerization

Production-ready Docker setup with multi-stage build, resource constraints (2 CPU cores, 1GB RAM, 512MB reserved), persistent database volume mounting, and Docker Compose orchestration. Optimized for lightweight deployment on constrained infrastructure.

03 System Architecture

Architecture Overview

The system follows an event-driven, asynchronous architecture pattern. Conversations are submitted through the REST API and immediately stored with a "pending" status. A background batch processor continuously polls for unprocessed conversations, sends them to Grok AI for analysis, and stores the structured insights back in the database. This decoupled design ensures fast API response times (202 Accepted) while handling AI processing asynchronously.

Data Flow - Conversation Submission

Step	Component	Action	Output
1	Client	POST /api/v1/conversations	HTTP Request
2	Rate Limiter	Token bucket check (100 req/s)	Allow / 429 Reject
3	Pydantic Schema	Validate request body	ConversationRequest
4	SQLAlchemy ORM	Insert to conversations table	status: pending
5	FastAPI	Return response	202 Accepted + conv_id

Data Flow - Background Processing

Step	Component	Action	Output
1	Batch Processor	Poll DB every 5 seconds	Pending conversations
2	Batch Processor	Update status to processing	Batch of 10
3	Grok Client	Send to Grok API (10 calls/s)	AI Analysis JSON
4	Grok Client	Parse & validate response	Structured insight data
5	SQLAlchemy ORM	Insert insight + update status	status: completed

Data Flow - Insights Retrieval

Step	Component	Action	Output
1	Client	GET /api/v1/insights?params	HTTP Request
2	FastAPI	Parse & validate query params	Typed parameters

3	SQLAlchemy ORM	Build filtered query	SQL with WHERE clauses
4	Database	Execute with indexes	Insight records
5	FastAPI	Serialize to JSON	InsightsResponse + metadata

Design Principles

• **Separation of Concerns** - API handling, data persistence, AI processing, and rate limiting are isolated into dedicated modules (main.py, models.py, grok_client.py, rate_limiter.py).

• **Async-First Design** - Every I/O operation uses async/await, from database queries (aiosqlite) to HTTP calls (httpx), maximizing throughput on limited CPU resources.

• **Graceful Degradation** - Individual conversation processing failures are isolated; the batch processor continues with remaining items. JSON parsing failures fall back to default values.

• **Resource Awareness** - SQLite chosen over PostgreSQL to minimize memory footprint. Batch sizes and concurrency limits are tuned for the 2-core, 1GB RAM constraint.

04 Technology Stack

The technology stack was carefully selected to balance performance, developer productivity, and resource efficiency within the 2-core CPU and 1GB RAM constraints.

Category	Technology	Version	Purpose
Web Framework	FastAPI	0.121.3	High-performance async API framework with auto-generated docs
ASGI Server	Uvicorn	0.38.0	Lightning-fast ASGI server for production deployment
Validation	Pydantic	2.12.4	Data validation and serialization with type hints
ORM	SQLAlchemy	2.0.36	Async ORM for database operations with 2.0 style
Database	SQLite + aiosqlite	0.20.0	Lightweight async database for resource-constrained env
HTTP Client	httpx	0.27.2	Modern async HTTP client for Grok API communication
Rate Limiting	Custom + slowapi	0.1.9	Token bucket and semaphore-based rate control
Containerization	Docker	3.8	Containerized deployment with resource limits
Language	Python	3.11+	Modern Python with async/await native support

Why FastAPI?

FastAPI was chosen as the web framework for several critical reasons: native async/await support for high-concurrency workloads, automatic OpenAPI documentation generation, Pydantic integration for request/response validation, dependency injection system for clean database session management, and exceptional performance benchmarks that rival Node.js and Go frameworks. Its type-hint-driven development approach also reduces bugs and improves code maintainability.

Why SQLite over PostgreSQL?

Given the resource constraints of 1GB RAM, SQLite was selected over PostgreSQL for several reasons: zero configuration and no separate server process required, minimal memory footprint (operates within the application process), sufficient performance for the expected workload, and simplified deployment with Docker volume mounting. The async wrapper (aiosqlite) ensures non-blocking I/O despite SQLite's inherently synchronous nature.

Why Grok AI?

xAI's Grok model provides state-of-the-art natural language understanding with a focus on Twitter/X platform content. Its API follows the OpenAI-compatible chat completions format, making it straightforward to integrate. The model excels at understanding social media language patterns,

informal text, and context-dependent sentiment, which is critical for accurate Twitter conversation analysis.

05 API Design

The API follows RESTful design principles with clear resource naming, proper HTTP method usage, appropriate status codes, and consistent error response formats.

Endpoint: POST /api/v1/conversations

Submits a new conversation for AI-powered analysis. Returns immediately with 202 Accepted status, indicating the conversation has been queued for background processing.

Request Body:

Field	Type	Required	Description
text	string	Yes	Conversation text content (min 1 char)
author	string	No	Author username or identifier
timestamp	ISO8601 datetime	No	Conversation timestamp (defaults to now)
raw_data	object	No	Additional metadata (stored as JSON)

Response Codes:

Status Code	Condition	Response Body
202 Accepted	Success	{"status": "accepted", "conversation_id": "conv_xxx", "messag
429 Too Many Requests	Rate limit exceeded	{"error": "rate_limit_exceeded", "retry_after": 5}
400 Bad Request	Invalid schema	{"error": "invalid_schema", "details": "..."}

Endpoint: GET /api/v1/insights

Retrieves analyzed insights with powerful filtering capabilities. Supports time-range queries, sentiment filtering, confidence thresholds, and pagination.

Query Parameters:

Parameter	Type	Required	Description
start_time	ISO8601 datetime	Yes	Start of time range filter
end_time	ISO8601 datetime	Yes	End of time range filter
limit	integer (1-1000)	No	Maximum results to return (default: 100)
min_confidence	float (0.0-1.0)	No	Minimum confidence score threshold
sentiment	enum	No	Filter: positive / negative / neutral

Response Structure:

The response includes an `insights` array containing analyzed conversations and a `metadata` object with pagination info (total_count, returned_count, start_time, end_time). Each insight contains the original text, conversation_id, timestamp, and a nested grok_analysis object with sentiment_score, clusters array, confidence score, and reasoning text.

Endpoint: GET /health

Simple health check endpoint returning `{"status": "ok"}`. Used for Docker health checks, load balancer probes, and monitoring systems.

06 Database Design

The database uses two core tables with a one-to-one relationship between conversations and their analysis results. Strategic indexing ensures fast queries even as data grows.

Conversations Table

Column	Type	Constraints	Description
id	String (PK)	Auto-generated	Format: conv_{uuid8} (e.g., conv_a1b2c3d4)
text	String	NOT NULL	Raw conversation text content
author	String	Nullable	Author username or identifier
timestamp	DateTime	NOT NULL, Indexed	Conversation timestamp
raw_data	JSON	Nullable	Original metadata stored as JSON
status	String	Indexed	pending | processing | completed | failed
created_at	DateTime	Auto-set	Record creation timestamp

Indexes: idx_conv_timestamp (timestamp), idx_conv_status (status)

Insights Table

Column	Type	Constraints	Description
id	Integer (PK)	Auto-increment	Sequential primary key
conversation_id	String (FK)	NOT NULL, Indexed	References conversations.id
timestamp	DateTime	NOT NULL, Indexed	Original conversation timestamp
text	String	NOT NULL	Conversation text (denormalized for fast access)
sentiment_score	Float	Indexed	Sentiment polarity: -1.0 to +1.0
clusters	JSON	Nullable	Topic categories array (e.g., ["praise", "product"])
confidence	Float	Indexed	Analysis confidence: 0.0 to 1.0
reasoning	String	Nullable	AI explanation of the analysis
grok_analysis	JSON	Nullable	Full Grok API response (for debugging)
created_at	DateTime	Auto-set	Record creation timestamp

Indexes: idx_insight_conversation_id, idx_insight_timestamp, idx_insight_sentiment, idx_insight_confidence

Indexing Strategy

Strategic indexing is applied to columns frequently used in WHERE clauses and JOIN conditions. The conversations table indexes `status` for efficient batch processor polling and `timestamp` for time-range queries. The insights table indexes `conversation_id` for relationship lookups, `timestamp` for range queries, `sentiment_score` for sentiment filtering, and `confidence` for threshold filtering.

07 Core Components

main.py - FastAPI Application

248 lines

The application entry point that initializes FastAPI, defines all API endpoints, manages the application lifecycle (startup/shutdown events), and coordinates between rate limiting, database, and batch processing components. Uses FastAPI's dependency injection for database session management and implements comprehensive error handling with proper HTTP status codes.

Key Implementation Details:

- Lifecycle management: Database initialization and batch processor start/stop
- Rate limiting integration at the endpoint level
- Pydantic model-based request/response validation
- Dynamic query building with SQLAlchemy for filtered insights retrieval
- Comprehensive error handling with rollback support

grok_client.py - Grok AI Client

139 lines

Handles all communication with xAI's Grok API. Implements structured prompt engineering for consistent JSON output, semaphore-based rate limiting (10 calls/second), retry logic with exponential backoff, and robust response parsing that handles markdown code block wrapping from the AI model.

Key Implementation Details:

- Prompt engineering for structured sentiment analysis and topic clustering
- Semaphore + sliding window rate limiting (10 calls/second)
- Markdown code block stripping from Grok responses
- Exponential backoff retry (3 attempts) for transient failures
- Graceful fallback with default values on JSON parsing errors
- 429 rate limit response handling with Retry-After header support

batch_processor.py - Background Processor

115 lines

An asyncio-based background task that continuously processes pending conversations. Uses asyncio.create_task for non-blocking execution, asyncio.gather for parallel batch processing, and proper status state machine management.

Key Implementation Details:

- Polling loop: checks for pending conversations every 5 seconds
- Batch processing: 10 conversations per cycle with parallel execution
- Status state machine: pending -> processing -> completed/failed
- Error isolation: individual failures don't affect the batch
- Graceful shutdown with task cancellation support

rate_limiter.py - Rate Limiting Engine

57 lines

Implements a token bucket algorithm with sliding window for precise rate limiting. Uses asyncio.Lock for thread-safe token management and deque for efficient time-window tracking. Provides Retry-After calculation for 429 responses.

Key Implementation Details:

- Token bucket algorithm with sliding time window
- Async lock for safe concurrent access
- Configurable max_requests and time_window parameters
- Retry-After header value calculation
- Global inbound_limiter instance (100 req/s)

models.py - Database Models

55 lines

SQLAlchemy 2.0 declarative models defining the database schema. Uses Column definitions with proper types, constraints, and strategic indexes for query performance.

Key Implementation Details:

- UUID-based conversation IDs (conv_{hex8} format)
- JSON columns for flexible metadata storage
- Strategic composite indexes on frequently queried columns
- Default value generators for timestamps and IDs

schemas.py - Pydantic Schemas

66 lines

Request/response validation schemas using Pydantic v2. Defines strict type validation, value ranges, and custom validators. Supports ORM mode for direct model serialization.

Key Implementation Details:

- ConversationRequest with custom text validator (non-empty, trimmed)
- GrokAnalysis with bounded float fields (-1.0 to 1.0, 0.0 to 1.0)
- SentimentFilter enum for type-safe query parameters
- InsightItem with from_attributes config for ORM compatibility

08 Rate Limiting & Performance

The system implements a dual-layer rate limiting strategy to protect both the API server and the external Grok AI service from overload.

Inbound Rate Limiter (Token Bucket)

The inbound rate limiter uses a **token bucket algorithm** with a sliding time window. It maintains a deque of request timestamps and removes entries older than the configured time window (1 second). If the deque length equals the max capacity (100), the request is rejected with a 429 status code and a calculated Retry-After header.

Parameter	Value	Description
Algorithm	Token Bucket	Sliding window with timestamp deque
Max Requests	100 / second	Maximum inbound API requests
Time Window	1.0 second	Sliding window duration
Concurrency	asyncio.Lock	Thread-safe async access
Rejection	HTTP 429	Includes Retry-After header

Grok API Rate Limiter (Semaphore + Window)

The Grok API rate limiter combines an **asyncio.Semaphore** (limiting to 10 concurrent calls) with a **time-window tracker** that maintains a list of recent call timestamps. If 10 calls have been made within the last second, the system calculates the precise wait time and sleeps before proceeding. This dual approach prevents both burst overload and sustained rate violation.

Parameter	Value	Description
Concurrency Limit	10 simultaneous	asyncio.Semaphore(10)
Rate Limit	10 calls / second	Time-window based throttling
Call Interval	100ms minimum	Effective spacing between calls
Retry Strategy	Exponential backoff	2^attempt seconds (max 3 retries)
429 Handling	Retry-After header	Respects server-specified wait time

Performance Optimization Strategies

- **Async I/O Everywhere** - All database operations use async SQLAlchemy with aiosqlite. All HTTP calls use async httpx. This maximizes CPU utilization during I/O wait times.

• **Batch Processing** - Instead of processing each conversation immediately, the system batches 10 at a time and processes them in parallel using asyncio.gather, reducing overhead.

• **Database Indexing** - Strategic indexes on timestamp, status, sentiment_score, and confidence columns ensure O(log n) query performance for filtered insight retrieval.

• **Connection Pooling** - SQLAlchemy's async engine manages a connection pool, avoiding the overhead of creating new database connections for each request.

• **Lightweight Stack** - SQLite eliminates the overhead of a separate database server process, keeping the total memory footprint well within the 1GB constraint.

• **Low-Temperature AI Calls** - Grok API calls use temperature=0.3 for more deterministic, consistent analysis results, reducing the need for retries due to inconsistent output.

09 Deployment

The application is containerized with Docker for consistent deployment across environments. Docker Compose provides orchestration with resource constraints matching production requirements.

Dockerfile Configuration

Stage	Command	Purpose
Base Image	`FROM python:3.11-slim`	Minimal Python image for small footprint
System Deps	`apt-get install gcc`	C compiler for native Python extensions
Dependencies	`pip install -r requirements.txt`	Install Python packages (cached layer)
Application	`COPY . .`	Copy application source code
Expose	`EXPOSE 8000`	Declare API port
Run	`uvicorn main:app --host 0.0.0.0 --port 8000`	Start ASGI server

Docker Compose - Resource Constraints

Resource	Limit	Reservation	Notes
CPU	2 cores	1 core	Sufficient for async workload + batch processing
Memory	1 GB	512 MB	SQLite in-process, no separate DB server
Storage	10 GB	-	Database file + application code
Port	8000:8000	-	Host-to-container port mapping
Restart	unless-stopped	-	Auto-restart on crash

Environment Variables

Variable	Required	Default	Description
`GROK_KEY`	Yes	-	xAI Grok API authentication key
`DATABASE_URL`	No	`sqlite+aiosqlite:///./insights.db`	Database connection string
`API_URL`	No	`http://localhost:8000`	Base URL for data ingestion script

10 Project Structure

The project follows a flat, modular structure where each file has a single, well-defined responsibility. This design simplifies navigation and testing.

File	Lines	Category	Responsibility
`main.py`	248	Application	FastAPI app, endpoints, lifecycle management
`models.py`	55	Data Layer	SQLAlchemy ORM models (Conversation, Insight)
`schemas.py`	66	Validation	Pydantic request/response schemas
`database.py`	37	Data Layer	Async engine, session factory, initialization
`grok_client.py`	139	AI Integration	Grok API client, prompt engineering, rate limiting
`rate_limiter.py`	57	Middleware	Token bucket rate limiter implementation
`batch_processor.py`	115	Processing	Background batch processing system
`ingest_data.py`	~100	Utility	Data ingestion script (sample + CSV)
`requirements.txt`	16	Config	Python dependency declarations
`Dockerfile`	24	DevOps	Container image build instructions
`docker-compose.yml`	24	DevOps	Container orchestration with resource limits
`test_api.sh`	~50	Testing	Shell-based API endpoint tests
`README.md`	241	Docs	Setup, API docs, usage, troubleshooting
`ARCHITECTURE.md`	208	Docs	Architecture documentation and design decisions
`TESTING.md`	-	Docs	Testing guide with scenarios and checklist

Total: ~8 Python source files, ~720+ lines of application code, 3 documentation files, 3 config/DevOps files

Module Dependency Graph

Module	Depends On
`main.py`	database, models, schemas, rate_limiter, batch_processor
`batch_processor.py`	models, grok_client, database
`grok_client.py`	(external: httpx, asyncio)
`rate_limiter.py`	(standalone: asyncio, collections)
`models.py`	(standalone: SQLAlchemy)
`schemas.py`	(standalone: Pydantic)
`database.py`	models

11 Challenges & Solutions

Grok API Response Inconsistency

Problem: Grok AI sometimes wraps JSON responses in markdown code blocks (```json...```) and occasionally returns malformed JSON, causing parsing failures.

Solution: Implemented a robust response parser that strips markdown code block wrappers (```json and ```) before JSON parsing. Added a 3-retry mechanism with exponential backoff (1s, 2s, 4s) and a graceful fallback that returns default values (sentiment_score=0.0, confidence=0.0, clusters=["unknown"]) when all retries fail.

Dual Rate Limiting Coordination

Problem: Needed to limit inbound API traffic (100 req/s) and outbound Grok calls (10 calls/s) with different strategies, while keeping the system responsive.

Solution: Implemented two separate rate limiting mechanisms: a token bucket algorithm with asyncio.Lock for inbound traffic (synchronous check on each request), and a semaphore + time-window tracker for Grok API calls (asynchronous wait with precise sleep calculation). The separation ensures neither limiter blocks the other.

Resource-Constrained Deployment

Problem: The application must run within 2 CPU cores and 1GB RAM, ruling out heavy database servers like PostgreSQL and limiting concurrency options.

Solution: Selected SQLite with async wrapper (aiosqlite) to eliminate separate database process overhead. Tuned batch sizes to 10 conversations and Grok concurrency to 10 calls to balance throughput with memory usage. Used Python 3.11-slim Docker image to minimize container size.

Asynchronous Background Processing

Problem: Background batch processing must run continuously without blocking API request handling, while sharing the same database and respecting rate limits.

Solution: Used asyncio.create_task to spawn the batch processor as a non-blocking background task. Each batch creates its own database session (AsyncSessionLocal) to avoid session conflicts. The processor uses asyncio.gather for parallel conversation processing within each batch, and individual failures are caught without affecting other conversations in the batch.

Graceful Error Isolation

Problem: A single Grok API failure or malformed response could potentially crash the entire batch processing pipeline, leaving conversations in a stuck state.

Solution: Implemented per-conversation error handling within _process_single. Failed conversations are marked with status="failed" while the batch continues. The batch processor's main loop also catches exceptions at the batch level and waits 5 seconds before retrying. This multi-level error handling ensures the system remains operational even under partial failure conditions.

Summary

The Insights Platform API demonstrates a production-grade approach to building AI-powered data analysis pipelines. By combining FastAPI's async capabilities with Grok AI's NLP power, the system achieves high throughput within strict resource constraints. The modular architecture, comprehensive error handling, and dual rate limiting strategy make it resilient and maintainable. This project showcases skills in API design, async programming, AI integration, database design, rate limiting algorithms, and containerized deployment.