NARRATIVE DETECTION MVP - IMPLEMENTATION ROADMAP

1. Setup & Tech Stack

What to do

Set up the development environment and infrastructure for the entire system.

How

Step 1: Install Core Dependencies

```
# Docker & Docker Compose
curl -fsSL https://get.docker.com | sh

# Ollama (Local LLM)
curl -fsSL https://ollama.com/install.sh | sh
ollama pull llama3.2:3b
```

Step 2: Initialize Project Structure

```
mkdir narrative-detection-mvp && cd narrative-detection-mvp
mkdir -p app/{models,detection,services,api,templates}
mkdir -p data/{raw_json,reports}
mkdir -p tests
```

Step 3: Create Configuration Files

- `docker-compose.yml` - Orchestrate PostgreSQL, Redis, API, Worker
- `requirements.txt` - Python dependencies
- `.env` - Environment variables
- `Dockerfile` - Container image for API and workers

Use (Tech Stack Explained)

Backend Core:

- **Python 3.11+**: Primary language
  - Why: Rich ML/NLP libraries, fast development
  - Fast enough for real-time processing
- **FastAPI**: Web framework for REST API
  - Why: Auto-generated docs, async support, type validation
  - Alternative considered: Flask (simpler but no async)

Data Storage:

- **PostgreSQL 16 + pgvector**: Primary database
    - Why: Stores tweets, users, narratives AND vector embeddings
    - pgvector extension: Enables similarity search without separate vector DB
    - Alternative considered: MongoDB (but we need ACID + vectors)
- **Redis**: Message queue + cache
    - Why: Queues your JSON data between scraper and processor
    - Also caches URL expansions and frequently accessed data
    - Alternative considered: RabbitMQ (more features but heavier)

Processing:

- **asyncio + threading**: Async workers
    - Why: Process multiple tweets concurrently
    - No need for Celery (overkill for MVP)

Frontend:

- **HTMX + Jinja2**: Dynamic UI without JavaScript framework
    - Why: Simpler than React, faster to build
    - Chart.js for visualizations
    - Alternative considered: React (too complex for MVP)

Infrastructure:

- **Docker Compose**: Local orchestration
    - Why: One command starts entire stack
    - Easy team collaboration (same environment)

Task Assignment

- **Person A (Usmaan & Muwafaq)**: Docker setup, database schema, environment config
- **Person D (Nadeem)**: Verify installations, document setup process

Deliverables

✅ All services running via `docker-compose up`
✅ Database accessible at `localhost:5432`
✅ Redis accessible at `localhost:6379`
✅ API returns health check at `localhost:8000/health`

---

2. Twitter Data Collector

What to build

Integration layer that consumes your existing 3-layer JSON output and feeds it into the system.

How

Architecture:

```
Your Scraper → JSON Files → Ingest Script → Redis Streams → Database
```

Step 1: Redis Streams Setup Create three streams for your three layers:

```
import redis

r = redis.Redis(host='localhost', port=6379)

# Add MICRO layer tweet
r.xadd('tweets:micro', {
    'layer': 'MICRO',
    'data': json.dumps(micro_json)
})

# Add MINUTE layer batch
r.xadd('tweets:minute', {
    'layer': 'MINUTE',
    'data': json.dumps(minute_json)
})

# Add HOURLY layer profile
r.xadd('tweets:hourly', {
    'layer': 'HOURLY',
    'data': json.dumps(hourly_json)
})
```

Step 2: File Watcher (Optional)

```
# watches data/raw_json/ folder
# automatically ingests new JSON files
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

class JSONHandler(FileSystemEventHandler):
    def on_created(self, event):
        if event.src_path.endswith('.json'):
            ingest_json_file(event.src_path)
```

Step 3: Manual Ingest Script

```
python scripts/ingest.py data/raw_json/micro_layer.json
python scripts/ingest.py data/raw_json/minute_layer.json
python scripts/ingest.py data/raw_json/hourly_layer.json
```

Use

Input Processing:

- **JSON parsing**: Python `json` module
  - Validates structure against expected schema
  - Handles malformed data gracefully
- **Redis Streams**: Message queue
  - **Why streams instead of pub/sub**: Persistence + consumer groups
  - Each layer gets its own stream
  - Workers consume from streams without blocking scrapers

Data Flow:

1. Your scraper outputs JSON to `data/raw_json/`
2. Ingest script reads JSON
3. Validates required fields (tweet_id, text, timestamp)
4. Pushes to appropriate Redis stream
5. Returns immediately (non-blocking)

Error Handling:

- Missing fields: Log warning, fill with NULL
- Duplicate tweet_id: Skip with log entry
- Malformed JSON: Save to `data/errors/` for manual review

Task Assignment

- **Person A (Usmaan & Muwafaq)**: Build ingest script, Redis integration, file watcher

Deliverables

✅ Script: `python ingest.py <json_file>` works
✅ Redis streams populated: Check with `redis-cli XLEN tweets:micro`
✅ Error handling logs issues without crashing

---

3. Preprocessing & Storage

What to build

Worker process that consumes from Redis, cleans data, and writes to PostgreSQL.

How

Worker Architecture:

```
# worker.py - Runs continuously
import redis
import psycopg2
from preprocessing import clean_tweet

r = redis.Redis()

while True:
    # Read from stream (blocks until data available)
    messages = r.xread({'tweets:micro': '$'}, count=10, block=5000)

    for stream, entries in messages:
        for msg_id, data in entries:
            tweet = json.loads(data['data'])

            # Clean and process
            processed = clean_tweet(tweet)

            # Save to PostgreSQL
            save_to_db(processed)

            # Acknowledge message
            r.xack('tweets:micro', 'worker-group', msg_id)
```

Cleaning Pipeline:

```
def clean_tweet(tweet):
    text = tweet['text_raw']

    # 1. Remove extra whitespace
    text = ' '.join(text.split())

    # 2. Remove/normalize emojis
    text_clean = text.encode('ascii', 'ignore').decode()

    # 3. Extract features
    hashtags = re.findall(r'#\w+', text)
    mentions = re.findall(r'@\w+', text)
    urls = re.findall(r'https?://[^\s]+', text)

    # 4. Generate text hash (for duplicate detection)
    normalized = ''.join(c.lower() for c in text if c.isalnum())
    text_hash = hashlib.md5(normalized.encode()).hexdigest()

    return {
        'id': tweet['tweet_id'],
        'text_raw': text,
        'text_clean': text_clean,
        'hashtags': hashtags,
        'mentions': mentions,
        'urls': urls,
        'text_hash': text_hash,
        'created_at': parse_timestamp(tweet['timestamp'])
    }
```

Use

Text Cleaning:

- **Python `re` module**: Regular expressions for pattern extraction
  - Extract hashtags: `#\w+`
  - Extract mentions: `@\w+`
  - Extract URLs: `https?://[^\s]+`
- **Unicode handling**: Remove emojis while preserving text
  - Method: `text.encode('ascii', 'ignore').decode()`
  - Alternative: `emoji` library for replacement

Feature Extraction:

- **Hashtags**: Store as PostgreSQL array `TEXT[]`
  - Enables queries: "Find all tweets with #scam"
- **Mentions**: Track interaction network
  - Later used for graph construction
- **URLs**: Critical for coordination detection
  - Many bots share the same malicious link

Duplicate Detection:

- **MD5 text hash**:
  - Normalize: lowercase + remove punctuation
  - Hash the normalized text
  - Same hash = exact duplicate (even if spacing differs)

Database Operations:

- **SQLAlchemy ORM**: Database abstraction

```
from sqlalchemy import create_engine
from models import Tweet

tweet = Tweet(
    id=processed['id'],
    text_raw=processed['text_raw'],
    text_clean=processed['text_clean'],
    hashtags=processed['hashtags']
)
session.add(tweet)
session.commit()
```

- **Batch inserts**: Process 100 tweets at once
  - Why: Reduces DB round-trips (10x faster)
  - Use: `session.bulk_insert_mappings()`

Timestamp Handling:

- **dateutil.parser**: Parse various date formats

- `from dateutil import parsertimestamp = parser.parse("2025-12-03T16:54:22.000Z")`
  - Handles ISO8601, RFC3339, Unix timestamps
  - Converts to UTC automatically

Task Assignment

- **Person A (Usmaan & Muwafaq)**: Build cleaning pipeline, database models
- **Person B (Omama & Hashir)**: Test feature extraction accuracy

Deliverables

✅ Worker consumes from Redis and writes to PostgreSQL
✅ All tweets have `text_clean`, `text_hash`, features extracted
✅ Batch processing: 100+ tweets/second throughput

---

4. Narrative Detection (Topics + Spikes)

What to build

Clustering algorithm that groups similar tweets into narratives and detects volume spikes.

How

Two-Step Process:

Step 1: Clustering (Group Similar Tweets)

```
from sentence_transformers import SentenceTransformer
from sklearn.cluster import HDBSCAN
import numpy as np

# Load embedding model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Get recent tweets
tweets = fetch_recent_tweets(hours=6)
texts = [t['text_clean'] for t in tweets]

# Generate embeddings
embeddings = model.encode(texts, show_progress_bar=True)

# Cluster
clusterer = HDBSCAN(
    min_cluster_size=5,       # Need at least 5 tweets
    min_samples=3,            # Density threshold
    metric='euclidean',
    cluster_selection_epsilon=0.5
)
clusters = clusterer.fit_predict(embeddings)
```

```
# Group tweets by cluster
narratives = {}
for idx, cluster_id in enumerate(clusters):
    if cluster_id == -1:  # Noise (unclustered)
        continue
    if cluster_id not in narratives:
        narratives[cluster_id] = []
    narratives[cluster_id].append(tweets[idx])
```

Step 2: Spike Detection (Volume Anomalies)

```
def detect_spike(narrative_id):
    # Get tweet timestamps for this narrative
    tweets = get_narrative_tweets(narrative_id)
    timestamps = [t['created_at'] for t in tweets]

    # Calculate baseline (past 24 hours)
    now = datetime.utcnow()
    baseline_start = now - timedelta(hours=24)
    baseline_tweets = [t for t in tweets if t['created_at'] >=
baseline_start]
    baseline_rate = len(baseline_tweets) / 24  # tweets per hour

    # Current rate (last hour)
    current_start = now - timedelta(hours=1)
    current_tweets = [t for t in tweets if t['created_at'] >=
current_start]
    current_rate = len(current_tweets)

    # Detect spike (3x threshold)
    velocity = current_rate / max(baseline_rate, 0.1)
    is_spike = velocity >= 3.0

    return {
        'is_spike': is_spike,
        'velocity': velocity,
        'current_rate': current_rate,
        'baseline_rate': baseline_rate
    }
```

Use

Embeddings (Semantic Vectors):

- **sentence-transformers**: Convert text → 384-dimensional vectors
    - Model: `all-MiniLM-L6-v2`
    - Why this model:
        - Fast (50 tweets/sec on CPU)
        - Good quality (outperforms TF-IDF)
        - Small download (80MB)
    - Alternative: `paraphrase-MiniLM-L6-v2` (slower but better quality)
- **Embedding properties**:
    - Similar meaning = similar vectors
    - Captures context (not just keywords)
    - Example: "5G causes cancer" similar to "5G health risks"

Clustering Algorithm:

- **HDBSCAN**: Hierarchical density-based clustering
  - Why not KMeans: We don't know # of narratives in advance
  - Why not DBSCAN: HDBSCAN better handles varying densities
  - Automatically finds number of clusters
  - Labels outliers as noise (cluster_id = -1)
- **Parameters explained**:
  - `min_cluster_size=5`: Need at least 5 tweets to form narrative
  - `min_samples=3`: 3 tweets needed in neighborhood for core point
  - `cluster_selection_epsilon=0.5`: Merge similar clusters

Storage:

- **pgvector**: Store embeddings in PostgreSQL
-
```
# Store embeddingtweet.embedding = embeddings[idx].tolist()   # Convert numpy to listsession.commit()# Later: Find similar tweetssimilar = session.query(Tweet).order_by( Tweet.embedding.cosine_distance(query_embedding)).limit(10).all()
```

Spike Detection Math:

- **Simple ratio-based**:
  - Velocity = (current hourly rate) / (24hr baseline rate)
  - Spike threshold: 3x (tunable parameter)
  - Example: 30 tweets/hour now vs 10 tweets/hour baseline = 3x = SPIKE

Temporal Bucketing:

- **pandas groupby**: Group tweets by time windows
-
```
import pandas as pddf = pd.DataFrame(tweets)df['time_bucket'] = df['created_at'].dt.floor('10min')volume = df.groupby('time_bucket').size()
```

Task Assignment

- **Person B (Omama & Hashir)**: Implement clustering, test on sample data
- **Person C (Omama & Hashir)**: Build spike detection logic

Deliverables

✅ Clustering groups similar tweets into narratives
✅ Each narrative has: title, summary, tweet_count, time_range
✅ Spike detector flags 3x+ volume increases
✅ API endpoint: `GET /narratives` returns list

---

5. Bot Detection (Basic Heuristics)

What to build

Rule-based scoring system that calculates bot probability for each account.

How

Score Calculation:

```
class BotDetector:
    def __init__(self):
        self.weights = {
            'posting_frequency': 0.30,
            'account_age': 0.25,
            'follower_ratio': 0.20,
            'repeat_text': 0.25
        }

    def score_user(self, user):
        score = 0.0

        # Feature 1: Posting frequency
        # Suspicious: >50 posts/day, Bot: >100
        posts_per_day = user['tweet_count'] / max(user['account_age_days'],
1)
        freq_score = min(posts_per_day / 100, 1.0)
        score += freq_score * self.weights['posting_frequency']

        # Feature 2: Account age (newer = suspicious)
        if user['account_age_days'] < 7:
            age_score = 1.0
        elif user['account_age_days'] < 30:
            age_score = 0.7
        elif user['account_age_days'] < 90:
            age_score = 0.3
        else:
            age_score = 0.0
        score += age_score * self.weights['account_age']

        # Feature 3: Follower ratio (anomalies)
        ratio = user['followers'] / max(user['following'], 1)
        if ratio < 0.1 or ratio > 10:
            ratio_score = 0.8
        elif ratio < 0.3 or ratio > 5:
            ratio_score = 0.5
        else:
            ratio_score = 0.0
        score += ratio_score * self.weights['follower_ratio']

        # Feature 4: Repeat text (get from DB)
        repeat_ratio = calculate_repeat_ratio(user['user_id'])
        repeat_score = min(repeat_ratio / 0.5, 1.0)
        score += repeat_score * self.weights['repeat_text']

        # Classify
        if score >= 0.7:
            label = 'BOT'
```

```
        elif score >= 0.4:
            label = 'SUSPICIOUS'
        else:
            label = 'ORGANIC'

        return score, label

def calculate_repeat_ratio(user_id):
    # Get all tweets from user
    tweets = get_user_tweets(user_id)

    # Count unique text hashes
    unique_hashes = set(t['text_hash'] for t in tweets)

    # Ratio of repeated content
    return 1 - (len(unique_hashes) / len(tweets))
```

Use

Feature Engineering:

- **Posts per day**: `tweet_count / account_age_days`
  - Calculate from HOURLY profile data
  - Bots often have 100+ posts/day
  - Stored in `users.posts_per_day` column
- **Account age**: `(today - account_created).days`
  - Parse join date from profile: "Joined August 2011"
  - Use `dateutil.parser` for flexible parsing
  - New accounts (<30 days) often used for campaigns
- **Follower ratio**: `followers / following`
  - Normal users: 0.5 - 2.0
  - Bots: Often <0.1 (follow many, few followers) or >10 (bought followers)
  - Handle division by zero: `max(following, 1)`
- **Repeat text ratio**: SQL query
- `WITH user_tweets AS (`
- `  SELECT text_hash FROM tweets WHERE user_id = ?`
- `)`
- `SELECT`
- `  COUNT(*) as total_tweets,`
- `  COUNT(DISTINCT text_hash) as unique_tweets,`
- `  1 - (COUNT(DISTINCT text_hash)::float / COUNT(*)) as repeat_ratio`
- `FROM user_tweets`

Scoring Formula:

- **Weighted sum**: Each feature contributes to final score
- **Weights** (sum = 1.0):
  - Posting frequency: 30% (most reliable indicator)
  - Account age: 25% (new accounts risky)
  - Follower ratio: 20% (good signal but can be gamed)
  - Repeat text: 25% (strong bot indicator)

Thresholds:

- Score 0.0-0.4: ORGANIC (normal user)
- Score 0.4-0.7: SUSPICIOUS (manual review)
- Score 0.7-1.0: BOT (high confidence)

Why This Approach:

- **Transparent**: Show users exactly why flagged
- **Tunable**: Adjust weights without retraining
- **Fast**: Pure math (1000 profiles/second)
- **No training data**: Works immediately

Update Schedule:

- Recalculate scores when:
  - New profile data arrives (HOURLY layer)
  - User posts 10+ new tweets
  - Manual request via API

Task Assignment

- **Person C (Omama & Hashir)**: Implement bot scoring, test on sample profiles

Deliverables

✅ Bot scores calculated for all users
✅ Scores stored in `users.bot_score` column
✅ Labels (ORGANIC/SUSPICIOUS/BOT) assigned
✅ API endpoint: `GET /users/{handle}` shows bot score

---

6. Coordinated Attack + Cross-Account Similarity

What to build

Detection system that finds groups of accounts posting identical/similar content in tight time windows.

How

Sliding Window Algorithm:

```
class CoordinationDetector:
    def __init__(self, time_window_minutes=10, similarity_threshold=0.85):
        self.time_window = time_window_minutes
```

```
        self.similarity_threshold = similarity_threshold

    def detect_coordination(self, tweets):
        clusters = []

        # Group by text hash (exact duplicates)
        hash_groups = {}
        for tweet in tweets:
            h = tweet['text_hash']
            if h not in hash_groups:
                hash_groups[h] = []
            hash_groups[h].append(tweet)

        # Find coordinated groups
        for text_hash, group in hash_groups.items():
            if len(group) < 3:  # Need at least 3 accounts
                continue

            # Check time window
            timestamps = sorted([t['created_at'] for t in group])
            time_span = (timestamps[-1] - timestamps[0]).total_seconds() /
60

            if time_span <= self.time_window:
                # Get unique users
                users = list(set(t['user_id'] for t in group))

                if len(users) >= 3:  # At least 3 different accounts
                    clusters.append({
                        'text_hash': text_hash,
                        'users': users,
                        'tweet_ids': [t['id'] for t in group],
                        'time_span_minutes': time_span,
                        'sample_text': group[0]['text_clean']
                    })

        # Also check embedding similarity (catches paraphrased content)
        semantic_clusters = self.find_semantic_similarity(tweets)

        return clusters + semantic_clusters

    def find_semantic_similarity(self, tweets):
        """Find similar (not identical) content using embeddings"""
        from sklearn.metrics.pairwise import cosine_similarity

        # Get embeddings
        embeddings = np.array([t['embedding'] for t in tweets])

        # Calculate similarity matrix
        sim_matrix = cosine_similarity(embeddings)

        # Find highly similar groups
        clusters = []
        processed = set()

        for i in range(len(tweets)):
            if i in processed:
                continue

            # Find tweets similar to tweet i
```

```
            similar_idx = np.where(sim_matrix[i] >
self.similarity_threshold)[0]

            if len(similar_idx) >= 3:  # At least 3 similar tweets
                similar_tweets = [tweets[j] for j in similar_idx]

                # Check time window
                times = [t['created_at'] for t in similar_tweets]
                time_span = (max(times) - min(times)).total_seconds() / 60

                if time_span <= self.time_window:
                    users = list(set(t['user_id'] for t in similar_tweets))

                    if len(users) >= 3:
                        clusters.append({
                            'type': 'SEMANTIC',
                            'users': users,
                            'tweet_ids': [t['id'] for t in similar_tweets],
                            'avg_similarity':
float(np.mean(sim_matrix[i][similar_idx])),
                            'time_span_minutes': time_span
                        })
                        processed.update(similar_idx)

        return clusters
```

Use

Exact Duplicate Detection:

- **Text hashing**: MD5 of normalized text
  - Already calculated in preprocessing stage
  - Stored in `tweets.text_hash` column
  - SQL query: `SELECT * FROM tweets WHERE text_hash = ? GROUP BY user_id HAVING COUNT(*) >= 3`

Sliding Window:

- **Time-based grouping**: Group tweets within N minutes
- `# PostgreSQL query with time window`
- `SELECT text_hash, array_agg(user_id), array_agg(id)`
- `FROM tweets`
- `WHERE created_at > NOW() - INTERVAL '24 hours'`
- `GROUP BY domain`
- `HAVING COUNT(DISTINCT user_id) >= 5`
- `ORDER BY account_count DESC`
- **Coordination signal**: 5+ accounts linking to same obscure domain = suspicious

Integration with Tweets:

- **Database storage**: Update `tweets.expanded_urls` column
- `# After expansiontweet.expanded_urls = [result['domain'] for result in expanded_results]session.commit()`

Rate Limiting:

- Respect domain rate limits
- Use connection pooling
- Batch requests (don't expand URLs one-by-one)

Task Assignment

- **Person A (Usmaan & Muwafaq)**: Build URL expansion service
- **Person B (Omama & Hashir)**: Maintain suspicious domain list

Deliverables

✅ URL expander service with Redis caching
✅ All shortened URLs resolved to final destinations
✅ Suspicious domains flagged
✅ Shared domain analysis in coordination detection

---

8. Community Detection (Graph)

What to build

Social network graph that identifies clusters of accounts based on interactions and content similarity.

How

Graph Construction:

```
import networkx as nx

def build_interaction_graph(tweets, users):
    G = nx.Graph()

    # Add nodes (accounts)
    for user in users:
        G.add_node(
            user['user_id'],
            handle=user['handle'],
            bot_score=user['bot_score']
        )

    # Add edges (interactions)
    for tweet in tweets:
        # Retweet edges
        if tweet.get('retweeted_from'):
            G.add_edge(
                tweet['user_id'],
                tweet['retweeted_from'],
                edge_type='RETWEET',
                weight=1.0
```

```
            )

        # Reply edges
        if tweet.get('reply_to_user'):
            G.add_edge(
                tweet['user_id'],
                tweet['reply_to_user'],
                edge_type='REPLY',
                weight=0.8
            )

        # Mention edges
        for mention in tweet.get('mentions', []):
            if mention in users_dict:
                G.add_edge(
                    tweet['user_id'],
                    users_dict[mention],
                    edge_type='MENTION',
                    weight=0.5
                )

    # Add similarity edges (content-based)
    similarity_edges = find_similar_content_pairs(tweets)
    for user_a, user_b, similarity in similarity_edges:
        if G.has_edge(user_a, user_b):
            # Strengthen existing edge
            G[user_a][user_b]['weight'] += similarity
        else:
            G.add_edge(user_a, user_b, edge_type='SIMILAR',
weight=similarity)

    return G

def find_similar_content_pairs(tweets):
    """Find pairs of users posting similar content"""
    from sklearn.metrics.pairwise import cosine_similarity

    # Group by user
    user_embeddings = {}
    for tweet in tweets:
        uid = tweet['user_id']
        if uid not in user_embeddings:
            user_embeddings[uid] = []
        user_embeddings[uid].append(tweet['embedding'])

    # Average embeddings per user
    user_avg_embeddings = {
        uid: np.mean(embs, axis=0)
        for uid, embs in user_embeddings.items()
    }

    # Find similar pairs
    users = list(user_avg_embeddings.keys())
    embeddings = np.array([user_avg_embeddings[u] for u in users])
    sim_matrix = cosine_similarity(embeddings)

    pairs = []
    for i in range(len(users)):
        for j in range(i+1, len(users)):
            if sim_matrix[i][j] > 0.7:  # High similarity
                pairs.append((users[i], users[j], sim_matrix[i][j]))
```

```
    return pairs
```

Community Detection:

```
from networkx.algorithms import community

def detect_communities(G):
    # Louvain algorithm (fast, good quality)
    communities = community.louvain_communities(G, weight='weight')

    # Analyze each community
    results = []
    for idx, comm in enumerate(communities):
        members = list(comm)

        # Calculate community stats
        bot_scores = [G.nodes[n]['bot_score'] for n in members]
        avg_bot_score = np.mean(bot_scores)

        # Internal vs external edges
        internal_edges = G.subgraph(members).number_of_edges()
        external_edges = sum(1 for u in members for v in G.neighbors(u) if
v not in comm)

        # Classify community
        if avg_bot_score > 0.6:
            community_type = 'BOT_CLUSTER'
        elif len(members) > 20 and internal_edges / len(members) > 3:
            community_type = 'COORDINATED_GROUP'
        else:
            community_type = 'ORGANIC'

        results.append({
            'community_id': idx,
            'size': len(members),
            'members': members,
            'avg_bot_score': avg_bot_score,
            'internal_edges': internal_edges,
            'external_edges': external_edges,
            'type': community_type
        })

    return results
```

Use

Graph Library:

- **NetworkX**: Python graph library
  - Why: Easy to use, good for <10K nodes (sufficient for MVP)
  - Alternative: Neo4j (for production scale >100K nodes)
  - In-memory processing (fast for analysis)

Node Types:

- **User nodes**: Each account is a node

- Attributes: handle, bot_score, account_age
- Stored in graph memory, not separate DB

Edge Types (weighted):

- **RETWEET**: weight=1.0 (strong signal of agreement/amplification)
- **REPLY**: weight=0.8 (engagement)
- **MENTION**: weight=0.5 (weaker signal)
- **SIMILAR**: weight=0.0-1.0 (based on content similarity)

Why weighted edges:

- Community detection considers edge weights
- Stronger connections = more likely same community
- Captures both explicit (retweet) and implicit (similar content) relationships

Community Detection Algorithm:

- **Louvain method**: Fast modularity optimization

```
from networkx.algorithms import community

# Detect communities
communities = community.louvain_communities(
    G,
    weight='weight',  # Use edge weights
    resolution=1.0    # Default (higher = more communities)
)
```

- **How it works**:
  1. Start with each node in its own community
  2. Iteratively merge communities that increase modularity
  3. Modularity: (internal edges - expected random) / total edges
  4. Stops when no more improvement
- **Why Louvain**:
  - Fast: O(n log n) time
  - Good quality: Near-optimal modularity
  - Hierarchical: Can zoom into sub-communities

Alternative Algorithms:

- **Greedy modularity**: Simpler, almost as good
- **Label propagation**: Faster but lower quality
- **Girvan-Newman**: Slow but interpretable (removes edges)

Community Classification:

- **BOT_CLUSTER**: avg_bot_score > 0.6
  - High concentration of bot accounts
  - Likely coordinated bot network
- **COORDINATED_GROUP**: Many internal edges

    - High interconnectivity
    - Members frequently interact
- **ORGANIC**: Normal user groups
    - Natural clustering around topics

Graph Metrics:

- **Modularity**: How well-separated are communities
    - Value: -0.5 to 1.0 (higher = better separation)
    - Good: >0.3
- **Density**: edges / possible_edges
    - Bot networks: Often high density (everyone connects)
    - Organic: Lower density (selective connections)

Visualization (optional for MVP):

```
import matplotlib.pyplot as plt

pos = nx.spring_layout(G)  # Position nodes
colors = [G.nodes[n]['bot_score'] for n in G.nodes()]

nx.draw(G, pos,
        node_color=colors,
        cmap=plt.cm.RdYlGn_r,
        with_labels=False,
        node_size=50)
plt.savefig('graph.png')
```

Task Assignment

- **Person C (Omama & Hashir)**: Build graph construction, community detection
- **Person B (Omama & Hashir)**: Test on sample networks, visualize

Deliverables

✅ Interaction graph built from tweets
✅ Communities detected and classified
✅ Bot clusters identified
✅ Graph stored in `edges` table
✅ API endpoint: `GET /communities` lists clusters

---

9. Narrative Origin Identification

What to build

Timeline analysis that traces each narrative back to its earliest tweets (origin seeds).

How

Origin Detection:

```
def find_narrative_origin(narrative_id):
    # Get all tweets in narrative
    tweets = session.query(Tweet).filter(
        Tweet.narrative_id == narrative_id
    ).order_by(Tweet.created_at).all()

    if not tweets:
        return None

    # First N tweets are origin candidates
    origin_window_minutes = 30
    first_tweet_time = tweets[0].created_at
    cutoff_time = first_tweet_time +
timedelta(minutes=origin_window_minutes)

    origin_seeds = [
        t for t in tweets
        if t.created_at <= cutoff_time
    ]

    # Analyze origin characteristics
    origin_users = list(set(t.user_id for t in origin_seeds))
    origin_bot_scores = [
        get_bot_score(uid) for uid in origin_users
    ]

    # Build spread timeline
    timeline = build_spread_timeline(tweets)

    return {
        'first_tweet_id': tweets[0].id,
        'first_tweet_time': tweets[0].created_at,
        'origin_seeds': [
            {
                'tweet_id': t.id,
                'user_handle': t.handle,
                'bot_score': get_bot_score(t.user_id),
                'text': t.text_clean,
                'timestamp': t.created_at
            }
            for t in origin_seeds
        ],
        'origin_user_count': len(origin_users),
        'origin_bot_ratio': np.mean(origin_bot_scores),
        'spread_timeline': timeline
    }

def build_spread_timeline(tweets):
    """Group tweets into time buckets"""
    import pandas as pd

    df = pd.DataFrame([
        {'timestamp': t.created_at, 'id': t.id}
        for t in tweets
    ])
```

```
    # Group by 5-minute buckets
    df['time_bucket'] = df['timestamp'].dt.floor('5min')
    timeline = df.groupby('time_bucket').size().to_dict()

    # Convert to list format
    return [
        {
            'time': bucket.isoformat(),
            'tweet_count': count
        }
        for bucket, count in sorted(timeline.items())
    ]
```

Spread Velocity Analysis:

```
def calculate_spread_metrics(timeline):
    """Analyze how fast narrative spreads"""
    if len(timeline) < 2:
        return {}

    # Time to reach milestones
    cumulative = 0
    milestones = {}

    for bucket in timeline:
        cumulative += bucket['tweet_count']

        if 'first_10' not in milestones and cumulative >= 10:
            milestones['first_10'] = bucket['time']
        if 'first_50' not in milestones and cumulative >= 50:
            milestones['first_50'] = bucket['time']
        if 'first_100' not in milestones and cumulative >= 100:
            milestones['first_100'] = bucket['time']

    # Calculate velocity (tweets per hour)
    first_time = datetime.fromisoformat(timeline[0]['time'])
    last_time = datetime.fromisoformat(timeline[-1]['time'])
    duration_hours = (last_time - first_time).total_seconds() / 3600
    total_tweets = sum(b['tweet_count'] for b in timeline)
    velocity = total_tweets / max(duration_hours, 0.1)

    # Find peak
    peak_bucket = max(timeline, key=lambda b: b['tweet_count'])

    return {
        'total_tweets': total_tweets,
        'duration_hours': round(duration_hours, 2),
        'velocity': round(velocity, 2),  # tweets/hour
        'peak_time': peak_bucket['time'],
        'peak_volume': peak_bucket['tweet_count'],
        'milestones': milestones
    }
```

Use

Timeline Construction:

- **pandas groupby**: Efficient time bucketing
- `import pandas as pd`

-
- `df['time_bucket'] = pd.to_datetime(df['timestamp']).dt.floor('5min')`
- `counts = df.groupby('time_bucket').size()`
- **Bucket sizes**:
  - 5 minutes: Detailed view (use for fast-spreading narratives)
  - 10 minutes: Standard view (most narratives)
  - 1 hour: Long-term trends

Origin Seed Selection:

- **Time window approach**: First 30 minutes
  - Why 30 min: Captures initial propagation
  - Too short (5 min): Might miss early amplifiers
  - Too long (2 hours): Includes secondary spread
- **Alternative: First N tweets**: Take first 5-10 tweets
  - Simpler but less robust
  - Might miss simultaneous origins

Spread Patterns:

- **Viral (organic)**:
  - Slow start → exponential growth → plateau
  - Origin from high-follower accounts
  - Low bot ratio in origin
- **Coordinated (attack)**:
  - Sudden burst → sustained volume → drop
  - Multiple simultaneous origins
  - High bot ratio in origin (>0.6)

Temporal Analysis:

- **Velocity**: tweets per hour
  - Organic viral: 50-200 tweets/hour at peak
  - Coordinated: 500+ tweets/hour burst
- **Milestones**: Time to reach 10, 50, 100 tweets
  - Fast (organic viral): 10 tweets in <15 min
  - Fast (coordinated): 10 tweets in <5 min
  - Slow (organic trend): 10 tweets in hours

Database Queries:

```
-- Get origin tweets
SELECT * FROM tweets
WHERE narrative_id = ?
ORDER BY created_at ASC
LIMIT 10;

-- Timeline aggregation
SELECT
  date_trunc('minute', created_at, 5) as time_bucket,
```

```
  COUNT(*) as tweet_count
FROM tweets
WHERE narrative_id = ?
GROUP BY time_bucket
ORDER BY time_bucket;
```

Storage:

- **narratives table**: Store origin metadata
- `UPDATE narrativesSET   origin_tweet_ids = ARRAY[123, 456, 789],   origin_handles = ARRAY['@user1', '@user2'],  first_seen = '2025-12-03 16:54:22'WHERE narrative_id = ?`

Task Assignment

- **Person B (Omama & Hashir)**: Build origin detection, timeline analysis

Deliverables

✅ Origin tweets identified for each narrative
✅ Spread timeline with 5-minute buckets
✅ Velocity metrics calculated
✅ API endpoint: `GET /narratives/{id}/origin` returns origin analysis

---

10. Advisory Countermeasures

a) Narrative Risk Score

What to build

Weighted risk scoring formula that combines multiple signals into single score.

How

Risk Calculation:

```
class RiskScorer:
    def __init__(self):
        # Configurable weights (sum = 1.0)
        self.weights = {
            'bot_ratio': 0.30,
            'spike_velocity': 0.25,
            'coordination': 0.25,
            'suspicious_urls': 0.20
        }

    def calculate_risk(self, narrative):
        score = 0.0
        details = {}
```

```
        # 1. Bot ratio (0-1, higher = more bots)
        bot_ratio = narrative['bot_ratio']
        bot_contribution = bot_ratio * self.weights['bot_ratio']
        score += bot_contribution
        details['bot_ratio'] = {
            'value': bot_ratio,
            'contribution': bot_contribution,
            'interpretation': self.interpret_bot_ratio(bot_ratio)
        }

        # 2. Spike velocity (normalize to 0-1)
        velocity = narrative['spike_velocity']
        # 5x spike = max risk
        velocity_norm = min((velocity - 1) / 4, 1.0)
        velocity_contribution = velocity_norm *
self.weights['spike_velocity']
        score += velocity_contribution
        details['spike_velocity'] = {
            'value': velocity,
            'normalized': velocity_norm,
            'contribution': velocity_contribution
        }

        # 3. Coordination score (0-1)
        coord = narrative['coordination_score']
        coord_contribution = coord * self.weights['coordination']
        score += coord_contribution
        details['coordination'] = {
            'value': coord,
            'contribution': coord_contribution
        }

        # 4. Suspicious URLs (normalize)
        url_count = narrative['suspicious_url_count']
        # 5+ URLs = max risk
        url_norm = min(url_count / 5, 1.0)
        url_contribution = url_norm * self.weights['suspicious_urls']
        score += url_contribution
        details['suspicious_urls'] = {
            'count': url_count,
            'normalized': url_norm,
            'contribution': url_contribution
        }

        # Classify level
        if score >= 0.7:
            level = 'HIGH'
            urgency = 'CRITICAL'
        elif score >= 0.4:
            level = 'MEDIUM'
            urgency = 'MODERATE'
        else:
            level = 'LOW'
            urgency = 'ROUTINE'

        return {
            'risk_score': round(score, 3),
            'risk_level': level,
            'urgency': urgency,
            'breakdown': details
        }
```

```
def interpret_bot_ratio(self, ratio):
    if ratio >= 0.7:
        return "SEVERE: Highly automated campaign"
    elif ratio >= 0.4:
        return "MODERATE: Significant bot activity"
    else:
        return "LOW: Mostly organic accounts"
```

Use

Scoring Components:

1. Bot Ratio (30% weight)

- **Calculation**: bot_accounts / total_accounts
- **Source**: From bot detection (Section 5)
- **Query**:
- `SELECT   COUNT(*) FILTER (WHERE u.bot_score >= 0.7) / COUNT(*)::float as bot_ratioFROM tweets tJOIN users u ON t.user_id = u.user_idWHERE t.narrative_id = ?`

2. Spike Velocity (25% weight)

- **Calculation**: current_rate / baseline_rate
- **Source**: From spike detection (Section 4)
- **Normalization**: (velocity - 1) / 4
  - 1x = 0.0 (no spike)
  - 5x = 1.0 (max risk)
  - 10x = still 1.0 (capped)

3. Coordination Score (25% weight)

- **Calculation**: Percentage of tweets in coordinated clusters
- **Source**: From coordination detection (Section 6)
- **Query**:
- `SELECT   COUNT(*) FILTER (WHERE t.id = ANY(    SELECT unnest(tweet_ids) FROM coordination_clusters  )) / COUNT(*)::float as coord_scoreFROM tweets tWHERE t.narrative_id = ?`

4. Suspicious URLs (20% weight)

- **Calculation**: Count of flagged domains
- **Source**: From URL expansion (Section 7)
- **Normalization**: min(count / 5, 1.0)
  - 0 URLs = 0.0
  - 5+ URLs = 1.0

Threshold Calibration:

- **LOW (0.0-0.4)**: Monitor, no immediate action

- **MEDIUM (0.4-0.7)**: Investigate, prepare response
- **HIGH (0.7-1.0)**: Immediate action required

Why These Weights:

- Bot ratio (30%): Strong indicator of artificial campaigns
- Spike velocity (25%): Rapid spread often indicates coordination
- Coordination (25%): Direct evidence of organized activity
- URLs (20%): Lower weight (sometimes organic sharing)

Tunability:

- Weights stored in config file
- Easy to adjust without code changes
- Can A/B test different weight combinations

b) Response Timing Recommendation

What to build

Rule-based system that recommends when to respond.

How

Timing Logic:

```
class ResponseAdvisor:
    def recommend_timing(self, narrative):
        risk_level = narrative['risk_level']
        velocity = narrative['spike_velocity']
        volume = narrative['tweet_count']

        # Decision tree
        if risk_level == 'HIGH':
            if velocity >= 3.0:
                return {
                    'timing': 'IMMEDIATE',
                    'timeframe': '< 30 minutes',
                    'rationale': 'High-risk fast-spreading narrative
requires immediate containment',
                    'priority': 'P0'
                }
            else:
                return {
                    'timing': 'URGENT',
                    'timeframe': '< 2 hours',
                    'rationale': 'High-risk but slower spread allows brief
preparation',
                    'priority': 'P1'
                }

        elif risk_level == 'MEDIUM':
            if velocity >= 2.0:
                return {
```

```
                    'timing': 'DELAY',
                    'timeframe': '2-4 hours',
                    'rationale': 'Moderate risk, gather more data before
responding',
                    'priority': 'P2'
                }
            else:
                return {
                    'timing': 'MONITOR',
                    'timeframe': '6-12 hours',
                    'rationale': 'Watch for escalation before committing
response',
                    'priority': 'P3'
                }

        else:  # LOW risk
            return {
                'timing': 'MONITOR',
                'timeframe': '24 hours',
                'rationale': 'Low risk, continue monitoring without
response',
                'priority': 'P4'
            }
```

Use

Decision Matrix:

Risk Level	Velocity	Timing	Timeframe
HIGH	Fast (≥3x)	IMMEDIATE	<30 min
HIGH	Normal (<3x)	URGENT	<2 hours
MEDIUM	Fast (≥2x)	DELAY	2-4 hours
MEDIUM	Normal	MONITOR	6-12 hours
LOW	Any	MONITOR	24+ hours

Rationale:

- **IMMEDIATE**: Viral misinformation spreading rapidly
  - Example: "Bank XYZ collapsing" false rumor
  - Every minute counts to stop panic
- **URGENT**: Serious but not viral yet
  - Example: Coordinated bot attack starting
  - Time to prepare quality response
- **DELAY**: Worth responding but not urgent
  - Example: Moderate criticism gaining traction
  - Better to respond thoughtfully
- **MONITOR**: Watch but don't engage yet

- Example: Low-volume complaint
- Responding might amplify ("Streisand effect")

Additional Factors:

- Time of day: Immediate response harder at 3am
- Day of week: Weekends vs weekdays
- Media involvement: Journalists asking = more urgent

c) Context-Specific Reply Strategy

What to build

Classification system that determines appropriate response tone and approach.

How

Strategy Selection:

```
class ReplyStrategySelector:
    def select_strategy(self, narrative):
        # Classify narrative type
        narrative_type = self.classify_narrative(narrative)
        bot_ratio = narrative['bot_ratio']
        volume = narrative['tweet_count']

        strategies = {
            'SCAM': {
                'tone': 'FIRM',
                'approach': 'Call out fraud directly with evidence',
                'template': 'WARNING: This is a known scam. [Evidence
link]',
                'include_legal': True
            },
            'PANIC': {
                'tone': 'CALM',
                'approach': 'Reassure with facts and authoritative
sources',
                'template': 'We understand concerns. Here are the facts:
[...]',
                'include_expert': True
            },
            'DEFAMATION': {
                'tone': 'EVIDENCE_BASED',
                'approach': 'Correct with verifiable data and sources',
                'template': 'The facts: [Data]. Sources: [Links]',
                'include_sources': True
            },
            'CRITICISM': {
                'tone': 'POLITE',
                'approach': 'Acknowledge concern and provide context',
                'template': 'We hear you. Here\'s what we\'re doing:
[...]',
                'include_action': True
            },
```

```
            'COORDINATED_BOT': {
                'tone': 'TRANSPARENT',
                'approach': 'Expose coordination evidence',
                'template': 'We\'ve detected coordinated inauthentic
activity. [Details]',
                'include_evidence': True
            }
        }

        strategy = strategies.get(narrative_type, strategies['CRITICISM'])

        # Adjust for bot ratio
        if bot_ratio > 0.6:
            strategy['tone'] = 'TRANSPARENT'
            strategy['call_out_bots'] = True

        return strategy

    def classify_narrative(self, narrative):
        """Classify narrative based on content and patterns"""
        keywords = narrative.get('top_keywords', [])
        urls = narrative.get('top_domains', [])
        bot_ratio = narrative['bot_ratio']

        # Rule-based classification
        scam_indicators = ['invest', 'guarantee', 'double your', 'limited
time']
        panic_indicators = ['collapse', 'crisis', 'shutdown', 'failing']
        defamation_indicators = ['fraud', 'illegal', 'scandal', 'corrupt']

        text_sample = ' '.join(keywords).lower()

        if any(ind in text_sample for ind in scam_indicators):
            return 'SCAM'
        elif any(ind in text_sample for ind in panic_indicators):
            return 'PANIC'
        elif any(ind in text_sample for ind in defamation_indicators):
            return 'DEFAMATION'
        elif bot_ratio > 0.6:
            return 'COORDINATED_BOT'
        else:
            return 'CRITICISM'
```

LLM Integration (Ollama):

```
import ollama

def generate_reply(narrative, strategy):
    prompt = f"""You are a social media response specialist. Generate a
brief, professional reply to address this narrative.

Narrative Summary: {narrative['summary']}
Tone: {strategy['tone']}
Approach: {strategy['approach']}

Key facts to include:
- Origin: {narrative['first_seen']}
- Volume: {narrative['tweet_count']} tweets
- Bot involvement: {narrative['bot_ratio']*100:.1f}%
```

```
Generate a reply that:
1. Matches the {strategy['tone']} tone
2. Is under 280 characters (Twitter limit)
3. {strategy['approach']}

Reply:"""

    response = ollama.generate(
        model='llama3.2:3b',
        prompt=prompt,
        options={
            'temperature': 0.7,
            'max_tokens': 150
        }
    )

    suggested_reply = response['response'].strip()

    return {
        'suggested_reply': suggested_reply,
        'tone': strategy['tone'],
        'approach': strategy['approach'],
        'requires_approval': True,  # Always require human review
        'template_used': strategy.get('template')
    }
```

Use

Narrative Classification:

- **Keyword matching**: Simple but effective
  - Scam: "invest", "guarantee", "limited"
  - Panic: "collapse", "crisis", "failing"
  - Use `any(keyword in text for keyword in indicators)`

Tone Selection:

- **FIRM**: For scams, fraud
  - Direct language, no hedging
  - Include legal warnings
- **CALM**: For panic/fear narratives
  - Reassuring language
  - Cite authoritative sources
- **EVIDENCE_BASED**: For false claims
  - Lead with data
  - Link to verifiable sources
- **POLITE**: For legitimate criticism
  - Acknowledge validity
  - Explain actions taken
- **TRANSPARENT**: For bot attacks
  - Show detection evidence
  - Call out inauthenticity

LLM (Ollama) Integration:

- **Model**: llama3.2:3b
    - Why: Fast (< 3 seconds), runs locally
    - Alternative: llama3:8b (better quality, slower)
- **Prompt engineering**:
    - Structured prompt with clear instructions
    - Include narrative context and tone guidance
    - Character limit constraint (Twitter)
- **Temperature**: 0.7
    - Not too creative (0.9+) = might hallucinate
    - Not too boring (0.1) = repetitive

Human-in-Loop:

- **Always require approval**: LLMs can make mistakes
- Show suggested reply in dashboard
- Allow editing before posting
- Log all approved/rejected suggestions

d) Evidence Summary Packet

What to build

HTML/Markdown report generator that bundles all evidence.

How

Report Generation:

```
from jinja2 import Template
import matplotlib.pyplot as plt

def generate_evidence_report(narrative_id):
    # Gather all data
    narrative = get_narrative(narrative_id)
    origin = get_origin_analysis(narrative_id)
    coordination = get_coordination_clusters(narrative_id)
    risk = calculate_risk(narrative)
    strategy = select_strategy(narrative)

    # Generate timeline chart
    timeline_chart = create_timeline_chart(narrative)

    # Build report data
    report_data = {
        'narrative': narrative,
        'origin': origin,
        'coordination': coordination,
        'risk': risk,
        'strategy': strategy,
        'timeline_chart': timeline_chart,
```

```
        'generated_at': datetime.utcnow().isoformat(),
        'report_id': f"RPT-{narrative_id}-{int(time.time())}"
    }

    # Render HTML
    template = Template(REPORT_TEMPLATE)
    html = template.render(**report_data)

    # Save
    report_path = f"data/reports/{report_data['report_id']}.html"
    with open(report_path, 'w') as f:
        f.write(html)

    # Also '1 hour'
  GROUP BY text_hash
  HAVING COUNT(DISTINCT user_id) >= 3
    AND MAX(created_at) - MIN(created_at) < INTERVAL '10 minutes'
```

Semantic Similarity:

- **Cosine similarity**: Measure angle between embedding vectors
  - Value 0-1 (1 = identical, 0 = opposite)
  - Threshold 0.85 catches paraphrasing
  - Example: "5G causes cancer" vs "5G health dangers" = 0.89 similarity
- **pgvector query**:
- `# Find tweets similar to target`
- `similar = session.query(Tweet).filter(`
- `    Tweet.embedding.cosine_distance(target_embedding) < 0.15  # 1-0.85`
- `).all()`

Coordination Signals:

1. **Same text + same time** = Strong signal (likely coordinated)
2. **Similar text + same time** = Moderate signal (might be organic trend)
3. **Same text + different times** = Weak signal (could be copypasta)

Storage:

- **coordination_clusters table**: Store detected groups
- `INSERT INTO coordination_clusters (  user_ids, tweet_ids, shared_text_hash,   similarity_score, time_window_minutes, cluster_size) VALUES (  ARRAY[123, 456, 789],  ARRAY[111, 222, 333], 'abc123def',  0.92,  8,  3)`

Real-time vs Batch:

- **Real-time**: Check each incoming tweet against last 1 hour
- **Batch**: Hourly job scans last 24 hours for patterns
- Why both: Real-time for alerts, batch for deep analysis

Task Assignment

- **Person C (Omama & Hashir)**: Implement coordination detection
- **Person B (Omama & Hashir)**: Test semantic similarity accuracy

Deliverables

✅ Detects groups of 3+ accounts posting same content within 10 minutes
✅ Catches paraphrased content via embedding similarity
✅ Stores coordination clusters in database
✅ API endpoint: `GET /coordination` lists clusters

---

7. Expanded URLs

What to build

URL unshortening service that follows redirects and identifies shared malicious domains.

How

URL Expansion Logic:

```
import httpx
from urllib.parse import urlparse
import time

class URLExpander:
    def __init__(self):
        self.cache = {}  # Cache expanded URLs
        self.suspicious_domains = set()  # Known bad domains

    async def expand_url(self, short_url, timeout=5):
        # Check cache first
        if short_url in self.cache:
            return self.cache[short_url]

        try:
            async with httpx.AsyncClient(follow_redirects=True,
timeout=timeout) as client:
                response = await client.head(short_url)
                final_url = str(response.url)

                # Extract domain
                domain = urlparse(final_url).netloc

                # Store in cache
                self.cache[short_url] = {
                    'final_url': final_url,
                    'domain': domain,
                    'status_code': response.status_code,
                    'expanded_at': time.time()
```

```
                }

                return self.cache[short_url]

        except Exception as e:
            # URL unreachable or timeout
            return {
                'final_url': short_url,
                'domain': urlparse(short_url).netloc,
                'error': str(e)
            }

    def is_suspicious_domain(self, domain):
        """Check against known malicious domains"""
        # Check local blacklist
        if domain in self.suspicious_domains:
            return True

        # Check heuristics
        suspicious_patterns = [
            'bit.ly',  # URL shorteners (not inherently bad, but used in
campaigns)
            'tinyurl.com',
            '.tk',  # Free TLDs often used for scams
            '.ml',
            '.ga'
        ]

        return any(pattern in domain for pattern in suspicious_patterns)
```

Batch Processing:

```
async def process_urls_batch(tweets):
    expander = URLExpander()

    # Extract all URLs
    all_urls = []
    for tweet in tweets:
        all_urls.extend(tweet.get('urls', []))

    # Expand concurrently
    tasks = [expander.expand_url(url) for url in all_urls]
    expanded = await asyncio.gather(*tasks)

    # Find shared domains
    domain_counts = {}
    for result in expanded:
        domain = result['domain']
        domain_counts[domain] = domain_counts.get(domain, 0) + 1

    # Flag domains shared by many accounts
    suspicious = {
        domain: count
        for domain, count in domain_counts.items()
        if count >= 5  # 5+ accounts sharing same domain
    }

    return suspicious
```

Use

HTTP Client:

- **httpx**: Async HTTP library
  - Why not `requests`: httpx supports async (much faster for batch)
  - `follow_redirects=True`: Automatically follows 301/302
  - `timeout=5`: Don't wait forever for dead links

Caching Strategy:

- **Redis cache**: Store expanded URLs
- `import redis`
- `r = redis.Redis()`
-
- `# Cache for 7 days`
- `r.setex(f'url:{short_url}', 604800, json.dumps(expanded_url))`
-
- `# Check cache`
- `cached = r.get(f'url:{short_url}')`
- `if cached:`
- `    return json.loads(cached)`
- **Why cache**: Same URLs appear in many tweets
  - Reduces HTTP requests by 80%+
  - Faster response times

Domain Extraction:

- **urllib.parse**: Parse URL components
- `from urllib.parse import urlparseurl = "https://example.com/path?param=value"parsed = urlparse(url)domain = parsed.netloc  # "example.com"`

Suspicious Domain Detection:

- **Blacklist approach**: Maintain list of known bad domains
  - Update periodically from threat intelligence feeds
  - Store in `suspicious_domains` set (O(1) lookup)
- **Heuristic approach**: Pattern matching
  - Free TLDs: .tk, .ml, .ga, .cf (often used for scams)
  - URL shorteners: bit.ly, tinyurl (not bad, but used in campaigns)
  - Newly registered domains: Check domain age via WHOIS

Shared Domain Analysis:

- **Aggregation**: Count how many accounts share each domain
- `SELECT   unnest(expanded_urls) as domain,  COUNT(DISTINCT user_id) as account_countFROM tweetsWHERE created_at > NOW() - INTERVAL`