1. Core Infrastructure

- **Zookeeper**:
  - Manages Kafka cluster coordination and maintains configuration data.
  - Configured via `zookeeper.properties` (port `2182`, data directory).
- **Kafka**:
  - Acts as the central message broker for streaming network flow data.
  - Configured via `server.properties` (broker ID, listeners, log directories, Zookeeper connection).
  - Uses topic `network-flows` to distribute flow data to consumers.

---

2. Data Producer

- **Network Sniffer (`kafka_produce_capture.py`)**:
  - **Role**: Captures network traffic (via Scapy), aggregates packets into flows, and publishes flow metrics to Kafka.
  - **Key Features**:
    - Flow tracking (bidirectional, TCP/UDP/ICMP).
    - Kafka integration with retries and batching.
    - Metrics include packet counts, byte sizes, TCP flags, and flow duration.
  - **Input**: Network interface (e.g., `wlo1`).
  - **Output**: Kafka topic `network-flows`.

---

3. Data Consumers

3.1 Redis TimeSeries Consumer (`kafka_consumer_redis.py`):

- **Role**: Subscribes to Kafka, ingests flow data into Redis TimeSeries for real-time analytics.
- **Key Features**:
  - Stores metrics like total bytes, packets, TCP flags, and protocol-specific stats.
  - Downsampling and retention policies for time-series data (1s, 10s, 1m resolutions).
- **Storage**: Redis with TimeSeries module (via Docker).

3.2 PostgreSQL Consumer (`kafka_consume_db.py`):

- **Role**: Persists flow data into PostgreSQL for structured querying and long-term storage.
- **Key Features**:
  - Two tables: raw JSON (`network_flows`) and structured metrics (`flow_metrics`).
  - Indexes for fast querying on IPs, ports, and timestamps.
- **Schema**: Includes fields like `src_ip`, `dst_ip`, `total_bytes`, and `timestamp`.

3.3 FastAPI Consumer (`kafka_consume_api.py`):

- **Role**: Provides a real-time API to access the latest flow data.
- **Key Features**:
  - In-memory storage of recent flows (last 1,000 entries).
  - Endpoints: `/flows`, `/flows/latest`, `/flows/search`.
  - Enriches data with `total_bytes` for API responses.
- **Runtime**: ASGI server (Uvicorn) on port `8888`.

---

4. Supporting Components

- **Docker Compose**:
  - Orchestrates Zookeeper and Kafka containers.
  - Maps volumes for data persistence (`./kafka-data`).
- **PostgreSQL/Redis**:
  - External services (configured via CLI arguments).
  - PostgreSQL credentials: `network_admin/securepassword123`.
  - Redis runs via `redislabs/redistimeseries` Docker image.

Data Flow

1. **Producer → Kafka**: Network flows are captured and sent to Kafka.
2. **Kafka → Consumers**: All three consumers read from the same Kafka topic in parallel.
3. **Consumers → Storage/API**:
   - Redis: Optimized for time-series queries (e.g., traffic spikes).
   - PostgreSQL: Structured analytics (e.g., historical trends).
   - FastAPI: Real-time monitoring via HTTP.

Technologies Used

- **Messaging**: Apache Kafka (with Zookeeper).
- **Data Capture**: Scapy (packet sniffing).
- **Storage**: Redis TimeSeries, PostgreSQL.
- **API**: FastAPI (Python), Uvicorn.
- **Orchestration**: Docker Compose.

This architecture enables scalable network monitoring with real-time analytics, historical storage, and API access. Each component is decoupled, allowing independent scaling (e.g., adding more consumers or partitioning Kafka topics).

Technologies

Technology	Role
Apache Kafka	Distributed message broker for streaming network flow data between producers and consumers.
Apache Zookeeper	Manages Kafka cluster coordination, leader election, and configuration.
PostgreSQL	Relational database for structured storage of flow metrics (e.g., IPs, ports, protocols).
Redis (TimeSeries Module)	Time-series database for real-time analytics (e.g., traffic spikes, protocol trends).
ASGI (Uvicorn)	Asynchronous server protocol to run the FastAPI application efficiently.

Programming Languages

Language	Role
Python	Primary language for all components: packet capture, consumers, and API.

Libraries & Frameworks

Library/Framework	Role
Scapy	Captures and analyzes network packets; extracts flow metrics (TCP/UDP/ICMP).
Kafka-Python	Python client for producing/consuming Kafka messages.
psycopg2-binary	PostgreSQL adapter for Python; inserts flow data into structured tables.
Redis/redis-py-cluster	Python client for interacting with Redis TimeSeries.
FastAPI	Web framework for building real-time API endpoints to query flow data.
Pydantic	Validates API request/response data models in FastAPI.

NumPy	Computes statistical metrics (e.g., mean packet lengths) in flow data.
argparse	Parses command-line arguments for script configuration.
logging	Standard Python module for application logging and debugging.

Tools & Infrastructure

Tool	Role
Docker	Containerizes Kafka, Zookeeper, and Redis for isolated deployment.
Docker Compose	Orchestrates multi-container setup (Kafka + Zookeeper).
Kafka CLI Tools	Manages Kafka topics (e.g., `kafka-topics --create`).
PostgreSQL CLI (psql)	Interacts with the PostgreSQL database via terminal.
Uvicorn	ASGI server to run the FastAPI application.

Gunicorn	Optional production-grade server for FastAPI (mentioned in requirements).
venv	Creates Python virtual environments for dependency isolation.

Data Formats & Protocols

Format/Protocol	Role
JSON	Serializes flow data for Kafka messages and PostgreSQL JSONB storage.
HTTP	Protocol for FastAPI endpoints to serve flow data.
TCP/UDP/ICMP	Network protocols analyzed by Scapy during packet capture.

Key Interactions

1. **Scapy → Kafka-Python**: Captured packets are aggregated into flows and sent to Kafka.
2. **Kafka-Python → PostgreSQL/Redis**: Consumers ingest Kafka messages into databases.
3. **FastAPI → Redis/PostgreSQL**: API queries databases to serve real-time flow data.
4. **Docker Compose → Kafka/Zookeeper**: Simplifies deployment of messaging infrastructure.

This stack enables scalable network monitoring with real-time analytics, historical storage, and API access. Each component is modular, allowing independent scaling (e.g., adding more Kafka partitions or Redis instances).

End-to-End Workflow of the Application

---

1. Infrastructure Setup

1. **Start Zookeeper & Kafka**:
    - Use `docker-compose.yml` to launch Zookeeper and Kafka containers.
    - Kafka brokers are configured to use Zookeeper for cluster coordination.
    - Kafka topic `network-flows` is created with 3 partitions and replication factor 1.
2. **Initialize Databases**:
    - PostgreSQL: Create `netflows` database and tables via `kafka_consume_db.py`.
    - Redis: Start Redis TimeSeries Docker container (`redislabs/redistimeseries`).

---

2. Data Production

1. **Packet Capture**:
    - Run `kafka_produce_capture.py` with a network interface (e.g., `wlo1`).
    - Scapy captures live network traffic and groups packets into **bidirectional flows** (based on IPs, ports, protocol).
2. **Flow Aggregation**:
    - Track metrics per flow: duration, packets, bytes, TCP flags (SYN/FIN/RST), etc.
    - Flows expire after `120s` of inactivity or explicit termination (e.g., TCP RST/FIN).
3. **Publish to Kafka**:
    - Serialize flow metrics into JSON and send to Kafka topic `network-flows`.
    - Kafka producer batches messages for efficiency and handles retries on errors.

---

3. Data Consumption

3.1 PostgreSQL Consumer

- **Role**: Persist structured flow data for long-term analytics.
- **Process**:
    1. Subscribe to Kafka topic `network-flows`.
    2. For each message:
        - Insert raw JSON into `network_flows` table.
        - Parse and store structured metrics (IPs, ports, protocols) in `flow_metrics` table.
        - Create indexes for fast querying (e.g., `src_ip`, `timestamp`).

3.2 Redis TimeSeries Consumer

- **Role**: Enable real-time time-series analytics (e.g., traffic spikes).
- **Process**:
    1. Subscribe to Kafka topic `network-flows`.
    2. For each flow:
        - Store metrics (bytes, packets, TCP flags) in Redis TimeSeries with labels (IP, protocol).
        - Downsample data for retention (1s → 10s → 1m resolutions).
        - Create compaction rules for aggregation (e.g., average bytes per minute).

3.3 FastAPI Consumer

- **Role**: Serve real-time flow data via HTTP API.
- **Process**:
    1. Subscribe to Kafka topic `network-flows` (latest offset).
    2. Maintain in-memory buffer (`deque`) of the last 1,000 flows.
    3. Expose endpoints:
        - `/flows`: List recent flows (paginated).
        - `/flows/latest`: Latest flow entry.
        - `/flows/search`: Filter by IP/protocol.

---

4. Data Querying

1. **PostgreSQL**:
    - Use SQL queries to analyze historical data (e.g., "Top 10 source IPs by bytes").

      ```
      SELECT src_ip, SUM(total_bytes)
      FROM flow_metrics
      GROUP BY src_ip
      ORDER BY SUM(total_bytes) DESC
      LIMIT 10;
      ```
2. **Redis TimeSeries**:
    - Query time-series metrics (e.g., "Bytes per protocol in the last hour"):

      ```
      TS.RANGE ts:protocol_bytes - + AGGREGATION avg 3600000
      ```
3. **FastAPI**:
    - Access real-time data via HTTP:

      ```
      curl http://localhost:8888/flows/search?src_ip=192.168.1.10
      ```

---

5. Monitoring & Scaling

- **Kafka Monitoring**:
    - Use `kafka-console-consumer.sh` to debug message flow:

      ```
      kafka-console-consumer --bootstrap-server localhost:9092 --topic
      network-flows
      ```
- **Scaling**:
    - Increase Kafka partitions to parallelize consumer workloads.
    - Deploy consumers as microservices (e.g., Kubernetes pods).

1. Zookeeper & Kafka (Docker Compose)

Purpose:

- **Zookeeper**: Manages Kafka cluster coordination (broker election, topic configuration).
- **Kafka**: Acts as the central message bus for streaming network flow data.

Internal Logic:

- Defined in `docker-compose.yml`:
  - Zookeeper runs on port `2182` with persisted data in `./kafka-data/zookeeper`.
  - Kafka broker connects to Zookeeper and exposes port `9092`.
  - Topics are created manually (e.g., `network-flows`).

Integration:

- All producers/consumers connect to Kafka via `localhost:9092`.
- Kafka uses Zookeeper for cluster health checks and metadata storage.

---

2. Network Sniffer Producer (`kafka_produce_capture.py`)

Purpose: Capture network packets, aggregate flows, and publish to Kafka.

Internal Logic:

1. **Packet Capture**:
   - Uses `scapy` to sniff packets on a network interface (e.g., `wlo1`).
   - Classifies packets into **bidirectional flows** (sorted by IP/port/protocol).
2. **Flow Tracking**:
   - Maintains a dictionary of active flows (`self.flows`).
   - Tracks metrics: packets, bytes, TCP flags, timestamps.
   - Flows expire after `120s` or on TCP FIN/RST.
3. **Kafka Publishing**:
   - Serializes flow stats to JSON.
   - Uses `KafkaProducer` with batching (`linger_ms=100`) and retries.

Integration:

- Sends data to Kafka topic `network-flows`.
- Downstream consumers (PostgreSQL, Redis, FastAPI) subscribe to this topic.

---

3. PostgreSQL Consumer (`kafka_consume_db.py`)

Purpose: Persist flow data for structured analytics.

Internal Logic:

1. **Kafka Consumption**:
   - Uses `KafkaConsumer` in `group_id='postgres-consumer-group'`.
   - Processes messages in batches with `poll(timeout_ms=1000)`.
2. **Database Operations**:
   - **Raw Storage**: Inserts JSON flow data into `network_flows` (JSONB column).
   - **Structured Storage**: Parses JSON into `flow_metrics` (columns: `src_ip`, `total_bytes`, etc.).
   - **Indexes**: Created on `src_ip`, `dst_ip`, `timestamp` for fast querying.
3. **Retry Logic**:
   - Reconnects to PostgreSQL on failure (max 5 retries).

Integration:

- Connects to PostgreSQL using `psycopg2` (credentials from CLI args).
- Enables SQL-based analytics (e.g., "Top talkers by traffic volume").

---

4. Redis TimeSeries Consumer (`kafka_consumer_redis.py`)

Purpose: Enable real-time time-series analytics.

Internal Logic:

1. **Kafka Consumption**:
   - Subscribes to `network-flows` in `group_id='redis-timeseries-consumer-group'`.
2. **TimeSeries Storage**:
   - Uses Redis commands (`TS.ADD`, `TS.CREATE`, `TS.CREATERULE`).
   - Stores metrics:
     - `flow_bytes_total`: Total bytes per flow (labels: `src_ip`, `protocol`).
     - `tcp_syn_count`: SYN flags per source IP.
   - Downsampling rules aggregate data into 10-second and 1-minute buckets.
3. **Pipeline Optimization**:
   - Batches Redis operations for efficiency.

Integration:

- Connects to Redis via `redis-py`.
- Enables queries like "Bytes per protocol in the last 5 minutes" via Redis CLI.

---

5. FastAPI Consumer (`kafka_consume_api.py`)

Purpose: Provide real-time API access to flow data.

Internal Logic:

1. **Kafka Consumption**:
   - Runs in a background thread (`start_kafka_consumer`).
   - Maintains a rotating buffer (`deque(maxlen=1000)`) of recent flows.
2. **API Endpoints**:
   - `/flows`: Returns last `N` flows (paginated).
   - `/flows/search`: Filters flows by IP/protocol.
   - Enriches data with `total_bytes = fwd_bytes + bwd_bytes`.
3. **Concurrency**:
   - Uses ASGI server (`uvicorn`) for asynchronous request handling.

Integration:

- Subscribes to Kafka and serves HTTP requests on port `8888`.
- Integrates with frontend dashboards for real-time monitoring.

---

6. Supporting Components

Docker Compose

- **Role**: Deploy Kafka/Zookeeper with persistent storage.
- **Key Configs**:
  - Volume mounts for Kafka/Zookeeper data.
  - Environment variables for broker ID, listeners, and replication.

PostgreSQL Schema

- **Tables**:
  - `network_flows`: Raw JSON data.
  - `flow_metrics`: Structured columns (IPs, ports, bytes).
- **Indexes**: Optimized for common queries (e.g., `src_ip`).

Redis TimeSeries

- **Data Retention**:
  - Raw data: 1 hour (1-second resolution).

- Aggregated data: 1 day (10-second), 1 week (1-minute).

Component Interactions

1. **Producer → Kafka**:
   - Network flows are serialized to JSON and published to `network-flows`.
2. **Kafka → Consumers**:
   - PostgreSQL/Redis/FastAPI consumers read from the same topic in parallel.
   - Each consumer group processes messages independently.
3. **Data Storage → API**:
   - FastAPI serves data from its in-memory buffer (not directly from DBs).
   - PostgreSQL/Redis are queried separately for historical/analytical use cases.

Error Handling & Scaling

- **Kafka**: Retries on producer/consumer errors.
- **PostgreSQL/Redis**: Reconnection logic for database failures.
- **Scaling**:
  - Kafka: Add partitions to distribute load.
  - Consumers: Deploy multiple instances (e.g., Kubernetes pods).

This modular architecture ensures fault tolerance, real-time processing, and scalability. Each component can be upgraded or replaced independently (e.g., swapping Scapy for `libpcap`-based capture).