Lyftr AI — Backend Assignment
Containerized Webhook API

Objective

Build a production-style FastAPI service that:

- Ingests inbound WhatsApp-like messages exactly once.

- Validates a simple HMAC-based signature on /webhook.

- Exposes proper liveness/readiness probes.

- Provides a paginated/filterable /messages endpoint.

- Exposes a Prometheus-style /metrics endpoint.

- Exposes a simple analytical /stats endpoint.

- Uses 12-factor environment configs.

- Emits structured JSON logs.

- Runs via Docker Compose using SQLite for storage.

Functional Requirements

1. POST /webhook

Purpose: Ingest inbound "WhatsApp-like" messages exactly once.

Request:

JSON body:

```
{
  "message_id": "m1",
```

```
  "from": "+919876543210",

  "to": "+14155550100",

  "ts": "2025-01-15T10:00:00Z",

  "text": "Hello"

}
```

- **Headers:**

    - **Content-Type:** `application/json`

    - **X-Signature:** `<hex HMAC-SHA256 of raw request body using WEBHOOK_SECRET>`

Signature:

- **Environment variable:** `WEBHOOK_SECRET` (non-empty string).

- **Compute:**
  `signature = hex(HMAC_SHA256(secret=WEBHOOK_SECRET, message=<raw request body bytes>))`

- **Behavior:**

    - If `WEBHOOK_SECRET` is not set → startup should fail (or `/health/ready` never becomes 200).

    - If `X-Signature` header is missing or invalid:

        - Return 401 with JSON: `{"detail": "invalid signature"}`

        - Do not insert anything into DB.

        - Log an error event.

    - If `X-Signature` is valid: proceed to validation + insert.

Validation:

- **message_id:** non-empty string.

- **from** / **to:** Strings in E.164-like form (start with **+**, then digits only).

- **ts:** ISO-8601 UTC string with `Z` suffix, e.g. `2025-01-15T10:00:00Z`.

- **text:** optional string, max length 4096.

Invalid payload → 422 (FastAPI/Pydantic style) with a JSON error description, and no DB insert.

Persistence & Idempotency:

- SQLite table `messages` with uniqueness on `message_id`.

- First valid call for a given `message_id`:

    - Insert a row.

    - Return `200 {"status": "ok"}`.

- Subsequent calls with the same `message_id` and valid signature:

    - Must not insert a second row.

    - Must still return `200 {"status": "ok"}` (idempotent).

    - No stack traces; errors must be handled gracefully.

Response (on success):

```
{
  "status": "ok"
}
```

---

2. `GET /messages`

Purpose: List stored messages, with pagination and basic filters.

Query parameters:

- **limit** (optional, int):

  - Default: 50

  - Min: 1, Max: 100

- **offset** (optional, int):

  - Default: 0

  - Min: 0

- **from** (optional, string):

  - If present, filter by `from_msisdn` exact match.

- **since** (optional, string):

  - ISO-8601 UTC timestamp.

  - If present, only return messages with `ts >= since`.

- **q** (optional, string):

  - Free-text search in `text` (case-insensitive substring match is enough).

Ordering:

- **Deterministic ordering:** `ORDER BY ts ASC, message_id ASC`.

- **"Newest last" still holds.**

Response shape:

```
{
  "data": [
    {
      "message_id": "m2",
      "from": "+919876543210",
      "to": "+14155550100",
```

```
      "ts": "2025-01-15T09:00:00Z",

      "text": "Earlier"

    }

    // ...

  ],

  "total": 4,          // total rows matching filters, ignoring
limit/offset

  "limit": 2,

  "offset": 0

}
```

- `total` must reflect the total number of records for the given filter, not just the number in `data`.

---

3. GET /stats

Purpose: Provide simple message-level analytics.

Behavior: No query params needed for this assignment.

Compute:

```
{

  "total_messages": 123,

  "senders_count": 10,

  "messages_per_sender": [

    { "from": "+919876543210", "count": 50 },

    { "from": "+911234567890", "count": 30 }

    // top senders, up to 10; sorted by count desc

  ],

  "first_message_ts": "2025-01-10T09:00:00Z", // null if no messages

  "last_message_ts": "2025-01-15T10:00:00Z"   // null if no messages
```

```
}
```

- Implementation can use SQL queries or Python aggregation, but performance should be reasonable for a few thousand rows.

---

4. Health Probes

- **GET /health/live:**

  - Always return 200 once the app is running.

- **GET /health/ready:**

  - Return 200 only if:

    - DB is reachable and schema is applied.

    - `WEBHOOK_SECRET` is set.

  - Otherwise return 503.

---

5. GET /metrics [OPTIONAL]

Purpose: Expose Prometheus-style metrics.

Format: Text-based Prometheus exposition format.

Minimum required metrics:

- A counter for total HTTP requests, with at least labels for `path` and `status`, e.g.:

  ```
  http_requests_total{path="/webhook",status="200"} 15
  http_requests_total{path="/webhook",status="401"} 2
  ```

- A counter for webhook processing outcomes, e.g.:

  ```
  webhook_requests_total{result="created"} 10

  webhook_requests_total{result="duplicate"} 5

  webhook_requests_total{result="invalid_signature"} 2
  ```

```
webhook_requests_total{result="validation_error"} 1
```

- Some latency measurement (bucketed or simple), e.g. buckets:

```
request_latency_ms_bucket{le="100"} 20

request_latency_ms_bucket{le="500"} 25

request_latency_ms_bucket{le="+Inf"} 25

request_latency_ms_count 25
```

- Your exact metric names can differ, but:
    - They must be stable and documented in README.
    - The evaluation script will just check that `/metrics` returns 200 and includes:

        - at least one line starting with `http_requests_total`,

        - and one line starting with `webhook_requests_total`.

---

6. Structured JSON Logs

- One JSON line per request.

- Required log keys:

    - `ts` (server time, ISO-8601),

    - `level`,

    - `request_id` (unique per request),

    - `method`,

    - `path`,

    - `status`,

    - `latency_ms`.

- For `/webhook` requests, logs must also include:

- message_id (when present),

- dup (boolean),

- result ("created", "duplicate", "invalid_signature",
   "validation_error" etc.).

Logs must be valid JSON per line (good for jq / log aggregation).

Non-Functional Requirements

- **Tech:** Python + FastAPI (or similar modern async framework).

- **DB:** SQLite only; DB file must live under a Docker volume (e.g. /data/app.db).

- **Validation:** Pydantic (or equivalent). Bad JSON / schema → 422.

- **Idempotency:** Enforced via DB uniqueness (PRIMARY KEY (message_id)) plus graceful handling in app layer.

- **Config via env vars:**

   - DATABASE_URL (e.g. sqlite:////data/app.db).

   - LOG_LEVEL (INFO / DEBUG).

   - WEBHOOK_SECRET.

   - No hard-coded paths/secrets.

- **Containerization:**

   - Multi-stage Dockerfile with small runtime image.

   - Docker Compose config with api service on http://localhost:8000.

Minimal Data Model

```
CREATE TABLE IF NOT EXISTS messages (
  message_id TEXT PRIMARY KEY,
```

```
    from_msisdn TEXT NOT NULL,
    to_msisdn   TEXT NOT NULL,
    ts          TEXT NOT NULL,   -- ISO-8601 UTC string
    text        TEXT,
    created_at  TEXT NOT NULL    -- server time ISO-8601
);
-- Optional: schema_version table if you want migrations (not
required for evaluation).
```

Deliverables (Repo Structure)

```
/app
  main.py           # FastAPI app, middleware, routes
  models.py         # SQLite init
  storage.py        # DB operations
  logging_utils.py  # JSON logger
  metrics.py        # metrics helpers (optional)
  config.py         # env loading (optional)
/tests
  test_webhook.py   # valid insert, duplicate, signature cases
  test_messages.py  # pagination + filters
  test_stats.py     # stats correctness
Dockerfile
docker-compose.yml
Makefile
README.md
```

Makefile targets:

- make up → docker compose up -d --build

- make `down` → `docker compose down -v`

- make `logs` → `docker compose logs -f api`

- make `test` → run your tests (if any)

What We Will Actually Run (Evaluation Script Outline)

For each submission, we will:

1. Set env vars and start the stack

```
export WEBHOOK_SECRET="testsecret"
export DATABASE_URL="sqlite:////data/app.db"

make up
# or: docker compose up -d --build
sleep 10
```

2. Health checks

```
curl -sf http://localhost:8000/health/live  >/dev/null

curl -sf http://localhost:8000/health/ready >/dev/null
```

3. Webhook + Signature (E)

```
BODY='{"message_id":"m1","from":"+919876543210","to":"+14155550100"
,"ts":"2025-01-15T10:00:00Z","text":"Hello"}'


# Invalid signature → expect 401
curl -s -o /dev/null -w "%{http_code}" \
  -H "Content-Type: application/json" \
  -H "X-Signature: 123" \
  -d "$BODY" \
```

```
  http://localhost:8000/webhook
# expected: 401


# Compute valid signature
# (We'll use a small helper script — you don't need to provide
this)
```

 Then:

```
 VALID_SIG="<computed from HMAC(SECRET, BODY)>"


# Valid signature → 200, row inserted
curl -s -o /dev/null -w "%{http_code}" \
  -H "Content-Type: application/json" \
  -H "X-Signature: $VALID_SIG" \
  -d "$BODY" \
  http://localhost:8000/webhook
# expected: 200


# Duplicate with same body + sig → 200, but no new row
curl -s -o /dev/null -w "%{http_code}" \
  -H "Content-Type: application/json" \
  -H "X-Signature: $VALID_SIG" \
  -d "$BODY" \
  http://localhost:8000/webhook
# expected: 200
```

4. Seed more messages for `/messages` and `/stats`

We'll send a few more valid messages with different `message_id`, `from`, `ts`, `text` (all with valid signatures).

5. Check `/messages` pagination & filters

```
# Basic list
curl -s "http://localhost:8000/messages" | jq .
```

```
# limit+offset
curl -s "http://localhost:8000/messages?limit=2&offset=0" | jq
'.data | length'
```

```
# filter by from=
curl -s "http://localhost:8000/messages?from=+919876543210" | jq .
```

```
# filter by since= and q=
curl -s "http://localhost:8000/messages?since=2025-01-15T09:30:00Z"
| jq .
curl -s "http://localhost:8000/messages?q=Hello" | jq .
```

We'll verify that:

- `total` matches the number of rows for that filter.

- `limit`/`offset` echo back correctly.

- Ordering is `ts` asc, `message_id` asc.

6. Check `/stats`

```
curl -s "http://localhost:8000/stats" | jq .
```

We'll verify that:

- `total_messages` matches the number of inserted rows.

- `senders_count` is correct.

○   messages_per_sender entries sum up to total_messages (for the senders listed).

○   first_message_ts and last_message_ts are correct min/max.

7. Check /metrics [OPTIONAL]

```
curl -s "http://localhost:8000/metrics" | head
```

 We'll check:

○   HTTP 200.

○   Output contains at least:

■   A line starting with http_requests_total.

■   A line starting with webhook_requests_total.

8. Logs

```
docker compose logs api | head -n 20
```

 Quick visual check that:

○   Logs are valid JSON per line (we may pipe through jq).

○   /webhook logs include message_id and dup.

9. Shutdown

```
make down
```

Scoring (10 points)

Most of the scoring will come from the automated checks above.

● Core correctness (4 pts)

- ○ Health endpoints.

- ○ `/webhook` success + idempotency.

- ○ Basic `/messages` listing and ordering.

- Advanced endpoints (4 pts)

  - ○ HMAC signature behavior.

  - ○ `/messages` pagination + filters.

  - ○ `/stats` correctness.

- Observability & ops (1 pt)

  - ○ `/metrics` with required metrics. **[OPTIONAL]**

  - ○ JSON logs, request_id, result fields.

- Docs & hygiene (1 pt)

  - ○ README with:

    - How to run (`make up`, URLs).

    - How to hit endpoints.

    - Brief "Design decisions" section:

      - How you implemented HMAC verification.

      - How your pagination contract works.

      - How you defined `/stats` and metrics.

Constraints & Use of AI

- No external services beyond Docker and Python (no Redis, no Postgres, etc.).

- SQLite only.

- All configuration via environment variables.

- You may use coding assistants (Copilot, ChatGPT, Cursor, etc.).

- In README, add a short note: "Setup Used"
  e.g. "VSCode + Copilot + occasional ChatGPT prompts".

We care about:

- Whether the system actually runs under the above script.

- Whether the behavior matches the exact semantics specified.

- How you structure and explain your solution.

Submission

- Share a GitHub repository link.

- Email to **careers@lyftr.ai** with subject: Backend Assignment – [Your Name]