

Signal SupportBot on Oracle Cloud: Streaming Case Mining + RAG with Two-Stage Reply Gating

(Your Name)

February 7, 2026

1 Scope (What We Build, Exactly)

We build a Signal group bot that:

- (1) Uses a dedicated Signal number (bot identity) and can be added to any group.
- (2) Ingests every new message (text + images) and turns it into *cases*: a case is a problem + solution block mined from chat.
- (3) Uses RAG over cases (not raw chat) to answer repetitive questions.
- (4) Replies only when:
 - someone mentions the bot, or
 - two-stage LLM gating decides (i) this is a help request worth considering and (ii) the bot has enough evidence to answer.
- (5) Optionally bootstraps recent history via a temporary linked-device ingest flow.

We support **text + images only**. Images are converted to a lightweight JSON containing only: `observations[]` and `extracted_text`. We do **not** store image bytes. Image extraction is **context-aware**: we pass the user message text that the image was attached to (possibly empty) to focus on relevant details (e.g., error codes, UI labels) without inventing facts.

2 Infrastructure (Oracle Cloud Only)

Everything runs on Oracle Cloud Infrastructure (OCI):

- **OCI Compute VM**: Ubuntu, Docker, Docker Compose.
- **Oracle Database**: stores ledger, cases, and app metadata (Autonomous DB or DB System).
- **No object storage required** for the MVP (we store image-derived text only).

3 Containers (Only 3, as requested)

We run exactly three containers on one VM:

- (1) **signal-bot**: bot number runtime + HTTP API + background worker in the same container.
- (2) **signal-ingest**: history bootstrap runtime (separate Signal profile; started when needed).
- (3) **rag**: Chroma vector database server (HTTP).

Chroma deployment via Docker is standard and simple. [3] (We cite it as a known deployment method; the doc link is in Appendix references.)

4 Data Stores and What Goes Where

4.1 Oracle DB (Relational Truth)

We store:

- **Raw ledger:** every message as normalized text (including image-to-text JSON).
- **Per-group buffer:** rolling text used by the case-miner.
- **Cases:** structured problem/solution units + evidence pointers.
- **Admin-group mapping:** who added the bot, used for the optional history flow.

4.2 Chroma (Vector Search for RAG)

We store one document per case:

- ID: `case_id`
- Text: concatenation of `problem.title` + `problem.summary` + `solution.summary` + `tags`
- Metadata: `group_id`, `status`, evidence message IDs, timestamps
- Embedding: computed by `EMBEDDINGMODEL`

5 Models (Explicit)

We use Google Gemini models as follows (names align with Google AI docs):

- **Gemini 3 Pro Preview** (`MODEL_IMG`, `MODEL_RESPOND`, `MODEL_BLOCKS`): vision-capable model for image extraction, response synthesis, and history chunk mining (higher quality, multimodal).
- **Gemini 2.5 Flash-Lite** (`MODEL_DECISION`, `MODEL_EXTRACT`, `MODEL_CASE`): cheap gating + buffer extraction (fast, many calls, cost-efficient).

Gemini models are accessed via Google's OpenAI-compatible API endpoint. [1, 2]

6 DB Schema (Minimal, Practical)

6.1 Tables

- `raw_messages`(`message_id`, `group_id`, `ts`, `sender_hash`, `content_text`, `reply_to_id`, `created_at`)
- `buffers`(`group_id`, `buffer_text`, `updated_at`)
- `cases`(`case_id`, `group_id`, `status`, `problem_title`, `problem_summary`, `solution_summary`, `tags_json`, `created_at`, `updated_at`)
- `case_evidence`(`case_id`, `message_id`)
- `admins_groups`(`admin_id`, `group_id`, `created_at`)
- `history_tokens`(`token`, `admin_id`, `group_id`, `expires_at`, `used_at`)
- `jobs`(`job_id`, `type`, `payload_json`, `status`, `attempts`, `updated_at`)

7 API (Small and Enough)

All endpoints live in the `signal-bot` container (FastAPI).

- GET `/healthz`
- POST `/history/token` ▷ admin requests optional history bootstrap
- GET `/history/qr/{token\}` ▷ return PNG QR
- POST `/history/cases` ▷ `signal-ingest` posts mined cases here
- POST `/retrieve` ▷ `(group_id, query, k) → top-K cases from Chroma`

8 Core Algorithms (Numbered, In Execution Order)

8.1 Notation (Prompts, JSON Contracts, LLM Calls)

Each prompt identifier `P_*` denotes a template string listed in the Appendix. We write a JSON-only LLM call as:

$$J \leftarrow \text{LLM}_{\text{json}}(m, P, \mathcal{V}),$$

meaning: render prompt template P with variable map \mathcal{V} , call model m , and obtain a JSON-only string J that parses to the required JSON object. All calls validate that J contains *exactly* the required keys and types; otherwise the call is retried once and then treated as an error.

8.2 Algorithm 1: Normalize Image to Lightweight JSON

We do not keep the image. We only keep a small JSON payload.

Algorithm 1: `ImageToTextJSON(image_bytes, context_text)`

Input: Image bytes; `context_text` (the user message the image was attached to; may be empty)
Output: JSON string J with fields `observations[]` and `extracted_text`
// Vision-capable call. Prompt template is `P_img` (Appendix).
1 $J \leftarrow \text{LLM}_{\text{json}}(\text{MODEL_IMG}, P_{\text{img}}, \{\text{IMAGE}=\text{image_bytes}, \text{CONTEXT}=\text{context_text}\})$
2 Validate JSON schema: keys are exactly `observations[]` and `extracted_text`
3 return J

8.3 Algorithm 2: Ingest Live Signal Message (Ledger + Jobs)

Runs in `signal-bot` container while listening to Signal events.

Algorithm 2: `IngestMessage(event)`

Input: Signal event with `group_id`, `sender`, `ts`, `text`, `images[]`, `reply_to`
Output: None (side effects)
1 Compute `message_id` (stable identifier from Signal)
2 Compute `sender_hash`
3 Set `content_text` \leftarrow `text` (or empty)
4 **foreach** `image` in `images[]` **do**
5 $J \leftarrow \text{ImageToTextJSON}(\text{image}, \text{context_text}=\text{text})$
6 Append "[image]
7 n" + J to `content_text`
8 Insert row into `raw_messages`
9 Enqueue job `BUFFER_UPDATE` with payload `{group_id, message_id}`
10 Enqueue job `MAYBE_RESPOND` with payload `{group_id, message_id}`

8.4 Algorithm 3: Update Buffer and Mine One Solved Case

Runs in the background worker loop (inside `signal-bot` container).

Algorithm 3: BufferUpdate(group_id, message_id)

Input: group_id, message_id

Output: Updated buffer; optionally a new case

```
1 Load message m from raw_messages
2 Render line ← Format(sender_hash, ts, content_text, reply_to)
3 Load B ← buffers[group_id] (or empty)
4 Append line to B
5 R ← LLMjson(MODEL_EXTRACT, P_extract, {BUFFER_TEXT=B})
6 Parse R: {found, case_block, buffer_new}
7 if found = true then
8   case_id ← MakeCase(group_id, case_block)
9   Set B ← buffer_new
10 Store updated B back into buffers
```

8.5 Algorithm 4: Turn a Case Block into a RAG Case (DB + Chroma)

No junk score. A lightweight LLM decides whether it is real and solved enough.

Algorithm 4: MakeCase(group_id, case_block)

Input: group_id, case_block (raw extracted messages)

Output: case_id or None

```
1 R ← LLMjson(MODEL_CASE, P_case, {CASE_BLOCK_TEXT=case_block})
2 Parse R: {keep, status, problem_title, problem_summary, solution_summary,
  tags[], evidence_ids[]}
3 if keep = false then
4   return None
5 Insert into cases and case_evidence
6 Create doc_text := problem_title + problem_summary + solution_summary + tags
7 Compute embedding
  (EMBEDDINGMODEL) Upsert into Chroma collection with : ID = case_id, text =
  doc_text, metadata = {group_id, status, evidence_ids} return case_id
```

8.6 Algorithm 5: Decide Whether to Respond (Two-Stage Gate + Context)

This implements your “no threshold” approach using explicit LLM decisions.

Algorithm 5: MaybeRespond(group_id, message_id)

Input: group_id, message_id**Output:** Either send a message or do nothing

```
1 Load message m from raw_messages
2 Load last N messages from the same group as plain context C (e.g., N = 40)
3 force  $\leftarrow$  (m mentions the bot)
4 if force = false then
5   D  $\leftarrow$  LLMjson(MODEL_DECISION, P_decision, {MESSAGE=m, CONTEXT=C})
6   Parse D: {consider}
7   if consider = false then
8     return do nothing
9 CK  $\leftarrow$  RetrieveCases(group_id, query=m, k=K)  $\triangleright$  Chroma search filtered by
   group_id
10
11 R  $\leftarrow$  LLMjson(MODEL_RESPOND, P_respond, {MESSAGE=m, CONTEXT=C, CASES=CK})
12 Parse R: {respond, text, citations[]}
13 if respond = false then
14   return do nothing
15 Send text to Signal group; append compact citations line if present
```

8.7 Algorithm 6: Optional History Bootstrap (Linked Device)

Important: Signal history sync works for the *admin account* when linking a new device, not for the bot's separate number. So we use a temporary ingest profile that an admin links and later unlinks.

Algorithm 6: BootstrapHistory(admin_id, group_id)

Input: admin_id, group_id**Output:** Cases inserted into DB + Chroma

```
1 API creates short-lived token in history_tokens
2 Admin scans QR and links signal-ingest profile as a device to their account
3 signal-ingest syncs recent chats; it filters messages by group_id
4 Render filtered history into chunks C1..Cn with small overlap
5 foreach chunk Ci do
6   R  $\leftarrow$  LLMjson(MODEL_BLOCKS, P_blocks, {CHUNK_TEXT=Ci})
7   Parse R: {cases[]} where each element has {case_block}
8   For each returned case_block: call MakeCase(group_id, case_block)
9 Mark token used; tell admin to unlink the ingest device
```

9 Deployment (OCI VM + Docker Compose)

9.1 Compose File (3 Containers)

```
services:
  signal-bot:
    build: ./signal-bot
    env_file: .env
    ports: ["8000:8000"]
    volumes:
      - /var/lib/signal/bot:/var/lib/signal/bot
```

```

signal-ingest:
  build: ./signal-ingest
  env_file: .env
  volumes:
    - /var/lib/signal/ingest:/var/lib/signal/ingest

rag:
  image: chromadb/chroma:latest
  environment:
    - IS_PERSISTENT=TRUE
  volumes:
    - /var/lib/chroma:/chroma/chroma
  ports: ["8001:8000"]

```

9.2 Environment Variables (Short List)

```

# Oracle DB
ORACLE_DSN=...
ORACLE_USER=...
ORACLE_PASSWORD=...

# Google AI (Gemini)
GOOGLE_API_KEY=...
MODEL_IMG=gemini-3-pro-preview
MODEL_DECISION=gemini-2.5-flash-lite
MODEL_EXTRACT=gemini-2.5-flash-lite
MODEL_CASE=gemini-2.5-flash-lite
MODEL_RESPOND=gemini-3-pro-preview
MODEL_BLOCKS=gemini-3-pro-preview
EMBEDDING_MODEL=text-embedding-004

# Signal
SIGNAL_BOT_E164=+...
SIGNAL_BOT_STORAGE=/var/lib/signal/bot
SIGNAL_INGEST_BASE=/var/lib/signal/ingest

# Chroma
CHROMA_URL=http://rag:8000

```

10 Consistency Check (Is This Deployable and Does It Make Sense?)

- **Bot without history:** Works immediately by ingesting new messages and mining cases.
- **History:** A dedicated-number bot cannot obtain old group history from other users automatically. The linked-device bootstrap is the correct way to get recent history securely: it syncs to a linked device of the same admin account, then we mine cases and discard the device. This is feasible and avoids insecure DB exports.
- **RAG store:** Chroma is a straightforward container. Retrieval is constrained by `group_id` in metadata.
- **Keys-only RAG:** Using only the problem (key) without storing solution text makes answers worse. The system needs the solution summary (and evidence IDs) to produce a useful reply. We keep minimal value text: problem+solution summaries + tags.

- **Images:** Not storing images is fine because we rely on extracted text/observations for RAG and citations.
- **Compute:** All heavy work is LLM calls; VM CPU is enough for glue code and Signal I/O.

A Prompts (Explicit, JSON-Only Contracts)

All prompts enforce:

- output is valid JSON
- no extra keys
- no markdown
- do not hallucinate facts not present in inputs

A.1 P_img (used by Algorithm 1; vision model)

```
SYSTEM: You extract only factual text and observations from an image.
You may use the provided CONTEXT (a user message) to focus on details that matter.
Do not invent facts that are not visible in the image.
Return ONLY valid JSON with exactly these keys:
- observations: array of short strings (facts visible in the image)
- extracted_text: string (best-effort text found in the image)

USER:
CONTEXT (may be empty):
{CONTEXT}

TASK: Extract observations and text from the attached image. If unreadable, keep
      extracted_text empty but still return JSON.
```

A.2 P_extract (used by Algorithm 3; Gemini 2.5 Flash-Lite)

Goal: detect and remove exactly one solved case from buffer.

```
SYSTEM: You analyze chat buffer text and detect if a solved support case is present.
Return ONLY JSON with keys:
- found: boolean
- case_block: string (exact subset of messages from the buffer forming ONE solved case
  ; empty if found=false)
- buffer_new: string (original buffer with case_block removed; if found=false, return
  original buffer)

Rules:
- A "solved case" must include a clear problem and a clear resolution/answer.
- Ignore greetings and pure acknowledgements.
- If multiple cases exist, extract only the earliest complete solved case.

USER: BUFFER:
{BUFFER_TEXT}
```

A.3 P_case (used by Algorithm 4; Gemini 2.5 Flash-Lite)

```
SYSTEM: Turn a case block into a structured support case.
Return ONLY JSON with keys:
- keep: boolean (true only if this is a real support case)
- status: string ("solved" or "open")
- problem_title: string (4-10 words)
- problem_summary: string (2-5 lines, concrete)
- solution_summary: string (0-10 lines; required if solved)
- tags: array of 3-8 short strings
- evidence_ids: array of message IDs if present in the block, else empty

Rules:
- If solved is not clear, set keep=false.
- Do not invent steps; only summarize what is present.

USER: CASE_BLOCK:
{CASE_BLOCK_TEXT}
```

A.4 P_decision (used by Algorithm 5; Gemini 2.5 Flash-Lite)

```
SYSTEM: Decide whether a new message is worth considering for a bot response.
Return ONLY JSON with keys:
- consider: boolean

consider=true only if:
- the message is asking for help or clarification, AND
- it is not trivial junk (greetings, "ok", emoji-only), AND
- it is relevant to group support context.

USER:
MESSAGE:
{MESSAGE}

CONTEXT (last messages):
{CONTEXT}
```

A.5 P_respond (used by Algorithm 5; Gemini 3 Pro Preview)

```
SYSTEM: You decide whether to respond in the group, and draft the response if yes.
Return ONLY JSON with keys:
- respond: boolean
- text: string (empty if respond=false)
- citations: array of short strings (e.g., ["case:123", "msg:1700000123"])

Rules:
- respond=true only if you can answer using the retrieved cases and context.
- If unsure, set respond=false (do not guess).
- Keep the response short, actionable, and specific.
- If you respond, include 1-3 citations referencing relevant cases.

USER:
MESSAGE:
{MESSAGE}
```

```
CONTEXT (last messages):  
{CONTEXT}  
  
RETRIEVED CASES (top-K):  
{CASES}
```

A.6 P_blocks (used by Algorithm 6; Gemini 3 Pro Preview)

```
SYSTEM: From a long history chunk, extract solved support cases.  
Return ONLY JSON with key:  
- cases: array of objects, each with:  
  - case_block: string (raw messages subset)  
Do NOT return open/unresolved cases.  
  
Rules:  
- Each case_block must contain both problem and solution.  
- Ignore greetings and unrelated chatter.  
- Keep case_block as exact excerpts from the chunk.  
  
USER: HISTORY_CHUNK:  
{CHUNK_TEXT}
```

B References

References

- [1] Google AI: Gemini Models. *ai.google.dev/gemini-api/docs/models*. (Accessed Feb 2026).
- [2] Google AI: Gemini API Pricing. *ai.google.dev/gemini-api/docs/pricing*. (Accessed Feb 2026).
- [3] Chroma Docs: Deploy with Docker. *docs.trychroma.com/guides/deploy/docker*. (Accessed Dec 2025).