CX Auto-Triage API

An HTTP API implemented in Node js that integrates with Zendesk to automatically triage support tickets using OpenAI. It can also answer product questions directly through a Slack slash command or a simple HTTP endpoint.

- Public base URL:
  `https://auto-triage-service-726253627972.us-central1.run.app`

- Source code:
  `https://github.com/CaioCohen/cx-auto-triage`

Endpoints

- Create a ticket in Zendesk (testing helper)
  `POST /api/tickets`

- Triage one specific Zendesk ticket by id
  `POST /api/tickets/:id/triage`
  Optional: `?force=true` to re-triage a ticket already marked as AI handled

- Knowledge Base answer (HTTP)
  `POST /api/answers/ask`

- Slack slash command receiver
  `POST /slack/command`
  Command name used: `/ask_observe`

1. Overview and Approach

1.1 Problem framing

The goal is to reduce time-to-triage and improve consistency. Agents often need to read a ticket, decide if extra checks are needed, and then write a summary with categories, tags, priority, and a short internal note. The API automates this.

1.2 Fictional system, mock database, and product description

This API is evaluated against a small, fictional observability product so reviewers can see grounded behavior without any real data. The setup has two parts: a mock database and a short product description file.

What the product is

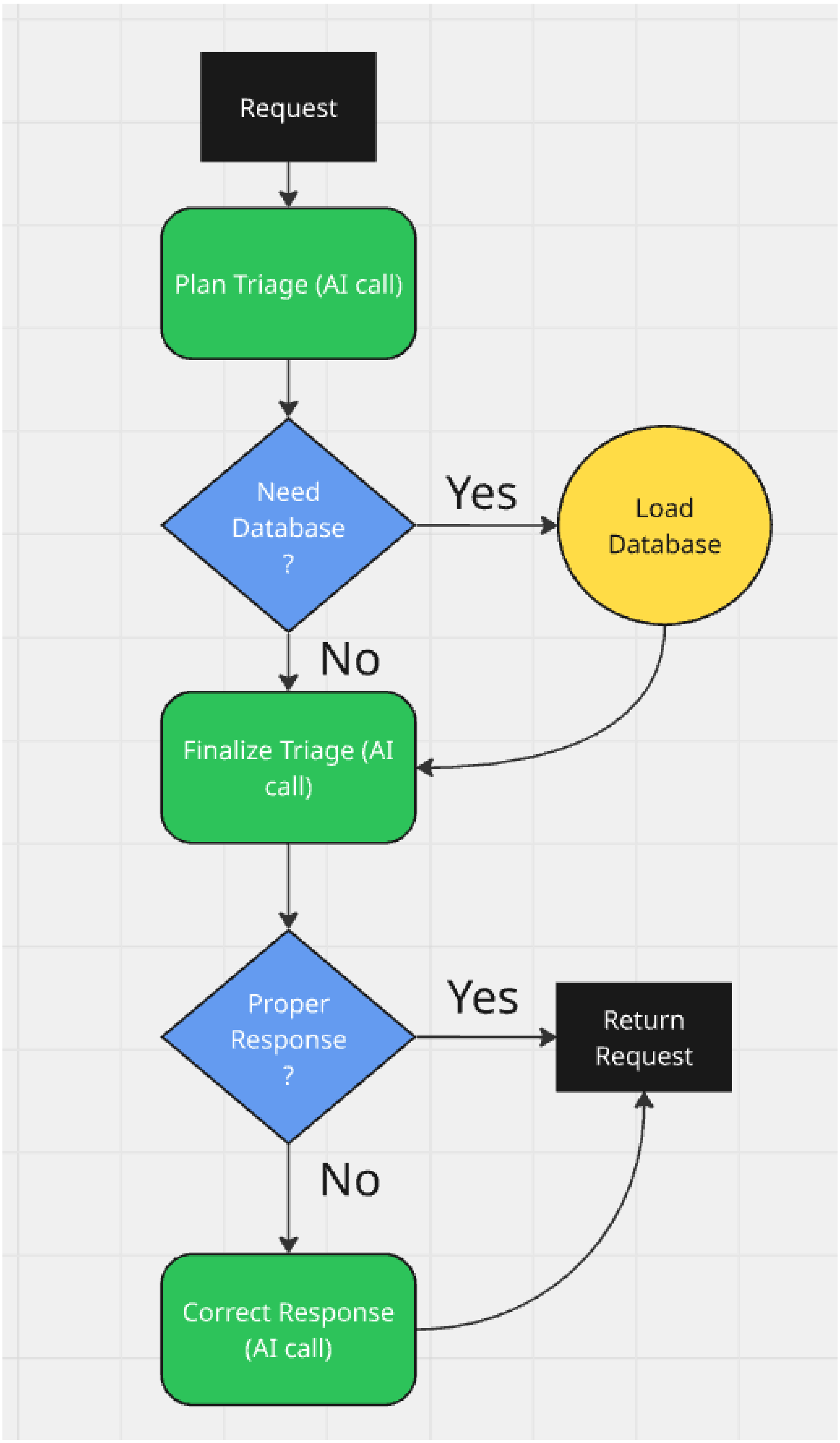
- A simple metrics and dashboards platform.
- Concepts: organizations, users, projects, dashboards, widgets, metrics, and recent metric samples.
- Typical questions: “is this user active”, “why is a widget hidden”, “does this metric have data”, “do I have access to this project”.

1.3 Architecture at a glance

```
src/
  controllers/        - request validation and orchestration
  repositories/       - Zendesk API integration
  routes/             - Express route definitions
  services/           - AI orchestration, KB answer, DB file upload
  data/               - mock_db.pdf (made up data)
  knowledge/          - description.txt (product rules and behavior)
  app.js, index.js    - Express bootstrap
```

Key dependencies: `express`, `axios`, `zod`, `openai`, `cors`.

1.4 Two-step triage with lightweight RAG



1. **Plan**
   `planTriage` uses the ticket's subject and body plus a product description file. It decides whether the model should consult the mock database. It also tries to extract literal values that appear in the ticket text only (email, org_id, project_name, dashboard_name, widget_title, metric_id) and sanitizes them to avoid hallucinations.

   ○ Model: `gpt-4o-mini` using `chat.completions` with JSON mode

   ○ Output: `{ need_db: "yes" | "no", ...extractedFields }`

   ○ Guardrails: strict JSON parsing with Zod, email must appear in the ticket text, `need_db` normalized

2. **Finalize**
   `finalizeTriage` produces the structured triage result. If `need_db` is yes, it attaches a PDF of the mock database as an `input_file` to the OpenAI Responses API so the model can ground its answer.

   ○ Model: `gpt-4o-mini` using `responses.create`

   ○ Input parts: system instructions, product description, ticket text, optional `input_file` containing the mock DB

Output schema enforced by Zod:

```
 {
  "category": "billing" | "bug" | "how_to" | "account" |
"feature_request" | "other",
  "priority": "low" | "normal" | "high" | "urgent",
  "language": "string(2..8)",
  "tags": ["... up to 10 ..."],
  "summary": "10..750 chars",
  "confidence": 0..1
}
```

   ○

   ○ One repair pass if the first attempt misses required fields

3. **Zendesk update**
   The controller merges tags safely, adds `ai_triaged` and a category tag (for example `cat_bug`), updates priority, and posts a private internal note summarizing the result. Tickets already marked with `ai_triaged` will be skipped unless `?force=true`.

1.5 RAG data and files

- `knowledge/description.txt` explains how the fictional observability product works at a high level. This gives the model product context.

- `data/mock_db.pdf` is a small, self-contained dataset that simulates orgs, users, projects, dashboards, widgets, metrics, and recent metric samples. The service `ensureDbFileId` uploads this PDF once to OpenAI's files API and caches its `file_id` for reuse.
  Reason for PDF: the Responses API accepts `input_file` of supported doc types. JSON is not supported directly.

1.6 Slack knowledge assistant

The slash command `/ask_observe` calls the same answer service used by the HTTP KB endpoint. The route sets `includeDb` based on simple keywords, and you can override with flags:

- `db:on Is the user elliot@acme.com active?`

- `db:off What is a widget?`

Slack replies are ephemeral. This is useful for quick answers about the product and the dataset without opening Zendesk.

2. Endpoints in detail

2.1 Create ticket

`POST /api/tickets`

Body shape:

```
{

  "requester": { "name": "Jane Customer", "email":
"jane@example.com" },

  "subject": "Cannot access account",

  "comment": { "body": "I get an error when logging in.", "public":
true },
  "priority": "high",
  "tags": ["login", "account"]
}
```

- Validated with Zod in `createTicketController`.
- Persists via `zendesk.repository.createTicket`.
- Returns `201` with `{ id, subject, status, url }`.

This endpoint exists to simplify testing and demoing. In a real deployment, Zendesk would create tickets through its own interfaces and web forms.

2.2 Triage one ticket

`POST /api/tickets/:id/triage`
Optional query: `?force=true`

Flow inside `triageOne`:

1. Validate id and read ticket from Zendesk.
2. Skip if already triaged unless `force=true`.
3. Call `planTriage`. If `need_db` is yes, call `ensureDbFileId` to upload or reuse the data file.
4. Call `finalizeTriage` with or without `fileId`.
5. Update the Zendesk ticket with merged tags, new priority, and a private internal note.

Return a compact response:

```
{ "id": 123, "status": "updated", "need_db": "yes", "category": "bug", "priority": "high" }
```

6.

2.3 Knowledge Base answer (HTTP)

`POST /api/answers/ask`

Body:

```
{ "query": "Is elliot@acme.com active?", "includeDb": true }
```

- If `includeDb` is true, the DB PDF is attached as `input_file`.
- Returns `{ "answer": "..." , "usedDb": true }`.

2.4 Slack command

`POST /slack/command`
Command used in Slack: `/ask_observe <question> [db:on|db:off]`

- Parses `text` from Slack.
- Simple heuristic for `includeDb`, override with flag `db:on` or `db:off`.
- Replies with a short, grounded answer.

3. Running the project

3.1 Use the published API (recommended for reviewers)

No setup needed. Use the base URL:

`https://auto-triage-service-726253627972.us-central1.run.app`

Examples:

```
# Health

curl -s
https://auto-triage-service-726253627972.us-central1.run.app/health



# Create a ticket (testing helper)

curl -s -X POST
https://auto-triage-service-726253627972.us-central1.run.app/api/tic
kets \
  -H "Content-Type: application/json" \
  -d '{
    "requester": {"name":"Jane Customer","email":"jane@example.com"},
    "subject":"Cannot access account",
    "comment":{"body":"I get an error when logging in.","public":true},
    "priority":"high",
    "tags":["login","account"]
  }'


# Triage a ticket by id
curl -s -X POST "https://auto-triage-service-726253627972.us-central1.run.app/api/tickets/123/triage?force=true"


# KB answer without Slack
curl -s -X POST https://auto-triage-service-726253627972.us-central1.run.app/api/answers/ask \
  -H "Content-Type: application/json" \
  -d '{"query":"Is elliot@acme.com active?","includeDb":true}'
```

3.2 Run locally

Requirements:

- Node 20 or newer
  If you must use an older Node, add `globalThis.File ??= (await import("node:buffer")).File;` at the very top of your entrypoint to support file

uploads.

- A Zendesk subdomain and API token for testing against your sandbox

Steps:

```
git clone https://github.com/CaioCohen/cx-auto-triage.git

cd cx-auto-triage

npm i
```

Create `.env` in the project root:

```
OPENAI_API_KEY=sk-...

ZENDESK_SUBDOMAIN=your_subdomain

ZENDESK_EMAIL=you@example.com

ZENDESK_API_TOKEN=...

KNOWLEDGE_PATH=./knowledge/description.txt

MOCK_DB_PATH=./data/mock_db.pdf

PORT=3000
```

Start:

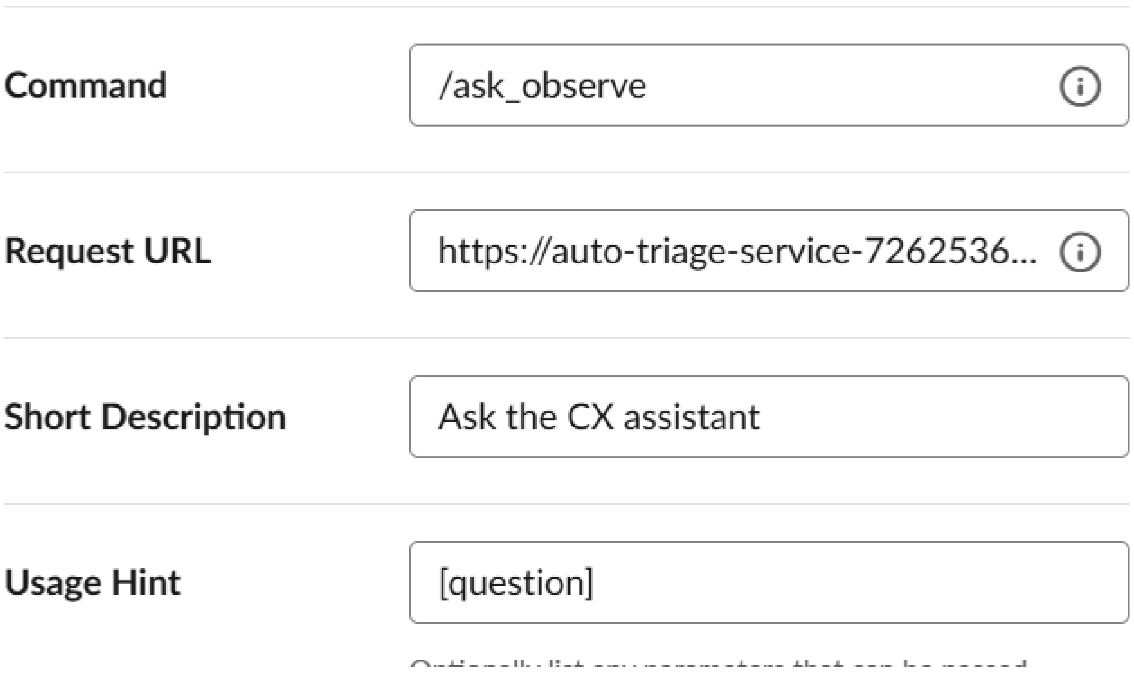
```
npm run start

# API on http://localhost:3000
```

Slack testing:

- LocalTunnel: You can use ngrok for creating a local tunnel for the localhost
- Or you could use the published url, to make things easier https://auto-triage-service-726253627972.us-central1.run.app/slack/command
- `Access` https://api.slack.com/apps `and create a new app`

- In the new app, go to Slash commands and create the new command, passing the above url



Command	/ask_observe
Request URL	https://auto-triage-service-7262536...
Short Description	Ask the CX assistant
Usage Hint	[question]

- inside OAuth & Permissions, click on install for <your work space>

4. Why this solves the problem

- **Fewer false positives, less back-and-forth**
  The model reads the ticket and decides if it actually needs data. Many issues are usage or configuration matters that can be answered immediately. When grounding is needed, it consults the product description and DB file to avoid guesses and catch inconsistencies early.

- **Faster first response and shorter handle time**
  The API produces a complete triage package in one call: category, priority, tags, and a clear internal summary. Agents start from a sensible baseline instead of a blank page, which reduces time to first action and time to resolution.

- **Consistent, auditable outcomes**
  Results follow a strict schema validated with Zod. Tags are merged safely and the ticket is marked with ai_triaged. The private note documents what was applied and why, so reviewers can audit decisions and improve prompts later.

- **Better customer experience**
  By filtering out simple or misrouted tickets automatically and flagging real issues quickly, customers get faster answers and fewer escalations. Complex cases arrive

to humans with context already assembled.

- **Self-service knowledge in Slack**
  The /ask_observe command answers common “how to” and “is X active” questions directly in chat. Teams get instant, grounded answers without opening Zendesk or hunting through docs, which reduces interruptions and ticket volume.

- **Operational safety valves**
  The system avoids double processing and allows force=true re-triage when needed. It is easy to extend with more checks, tighter prompts, auth, or Secret Manager without changing the public contract.

- **Cost and focus benefits**
  Routine triage is automated so human time can be spent on debugging, customer communication, and fixes. You pay for AI only when useful, since the plan step skips DB access for simple cases.

---

5. Security and scope notes

- This is a take-home project using fictional data. Authentication and authorization were intentionally out of scope. Endpoints do not require tokens and should not be exposed without a gateway in production.

- Secrets should be stored in a manager such as Google Secret Manager. Cloud Run supports mounting secrets as environment variables.

---

6. Limitations and future work

- Merge the 2 calls into a single structured tool call when `need_db` is often no, to reduce latency.

- Add a small retrieval layer for focused DB slices. Today the whole DB is passed as a PDF file to the Responses API. For larger datasets, use vector stores and file_search.

- Add an auth layer with scopes for triage and KB access.

- Add observability of model outputs and human correction loops to improve tags and category over time.

- Add async triage for bulk tickets with a small job queue and backoff to respect Zendesk rate limits.

7. Video Demonstrations

On the following links, you'll find two videos demonstrating how the API works, one for Slack command bot, and the other for triaging a Zendesk ticket:

Slack Command Bot

Zendesk Demonstration