15

The Capstone

A full-stack LLM app, end to end — small enough to hold in your head

Chapter 15 · Streamlit UI -> FastAPI -> Gemini -> SQLite

15 The journey — one app, five moves

We build a real product you can show a hiring manager. Four small files, one new library. Here are the five ideas we walk through, one at a time.

By the end you understand where the key lives, why the tiers split, and how a UI, an API, an LLM and a database run together.

15

What we are building

A Customer Feedback Analyzer

A business owner pastes a stack of reviews — one per line — clicks Analyze, and gets back a table. Each review comes back tagged with three fields.

label

positive,
negative,
or neutral

score

a number from
0.0 to 1.0

theme

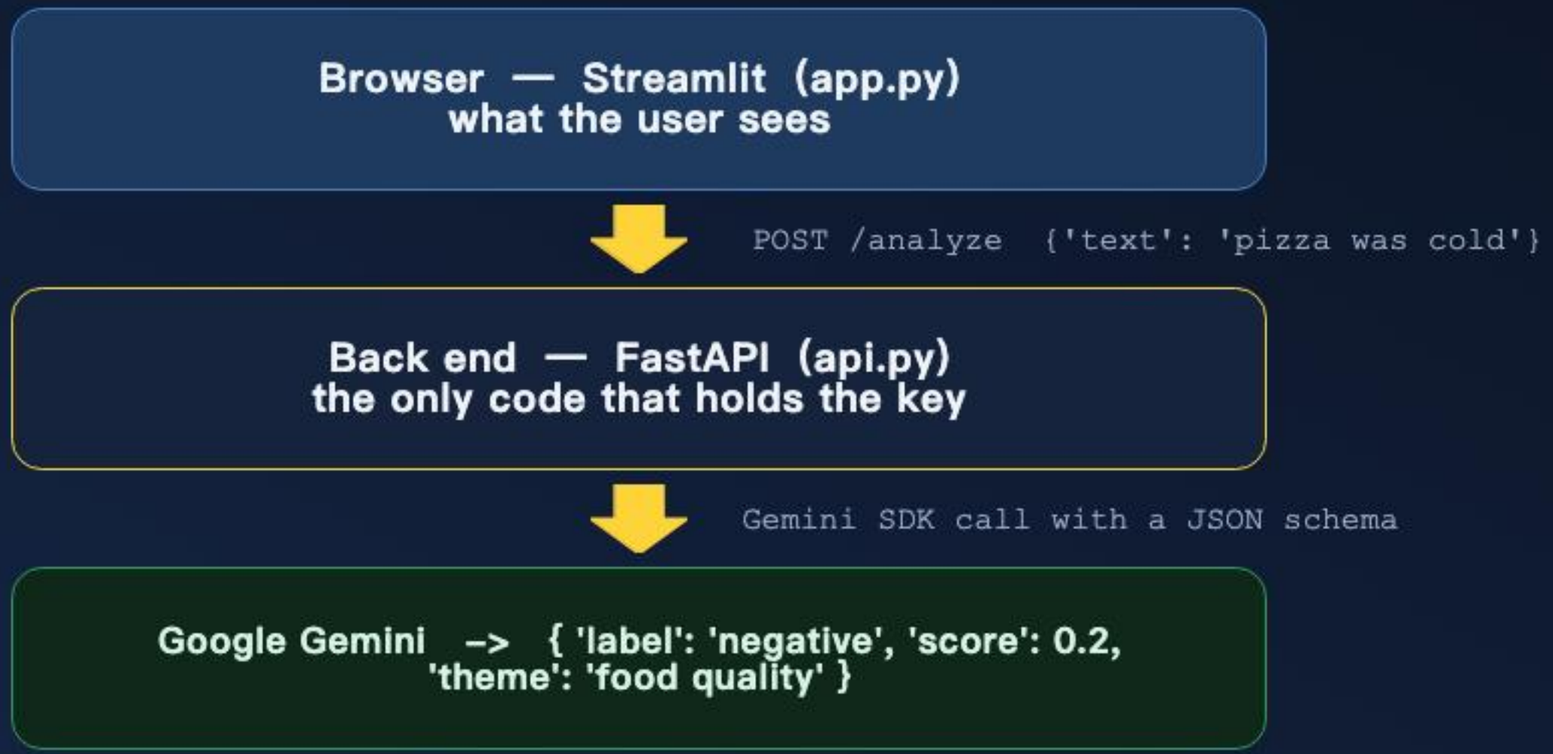
what it is about:
delivery, price,
service, food...

Above the table: how many reviews, the average score, the percent positive. A Save button stores the run; a Load view brings old runs back.

15

The architecture — three tiers, three files

Each hop is one arrow



The front end NEVER talks to Gemini directly — it only calls OUR back end. The result flows back to a table; a Save writes it to SQLite (database.py).

15

Why split them at all?

Streamlit could call Gemini directly — three reasons we don't

1. The key stays server-side

A Streamlit app is a web page. Anything the page can see, a curious visitor can see. A key in app.py would leak the moment you publish.

2. The API is reusable

Once /analyze exists, a mobile app, a cron job or a colleague's script can all use it. The logic isn't trapped inside one UI.

3. The API is testable

You can hit /analyze with a test tool and check its output — without ever opening a browser.

Reading and judging: the first question for any LLM app is — where does the API key live? It belongs ONLY in the back end. A Streamlit file that imports the model SDK and holds the key is a red flag.

15

The Gemini SDK — the one new library

An SDK is just the official library for talking to a service

Chapter 14 taught Streamlit, FastAPI and SQLite. The genuinely new piece here is Google's Gemini SDK (software development kit).

```
pip install google-genai        # or:  uv add google-genai
```

```
from google import genai
# note the unusual form: you import 'genai' FROM the 'google' namespace
```

Install it, then import it. The odd part is the import line — 'genai' comes out of the shared 'google' package, not a package called genai.

15

The key lives in a .env file

A plain list of NAME=value secret lines, loaded at runtime

```
GOOGLE_API_KEY=your-key-goes-here      # this line lives in .env
```

```
from dotenv import load_dotenv

load_dotenv()    # reads .env and puts each line into
the environment
```

A .env file is NOT Python — it is loaded at runtime. The Gemini SDK looks for GOOGLE_API_KEY automatically, so once load_dotenv() has run you never pass the key by hand.

Reading and judging: .env MUST be listed in .gitignore so it is never committed. A leaked key in git history is a real incident. If you clone a repo and see a tracked .env with a live key — that is a serious bug. Flag it, and rotate the key.

15

The basic call — three lines

Make a client, send a prompt, read the reply

```
client = genai.Client()                    # reads the key from the environment
resp = client.models.generate_content(
    model='gemini-2.5-flash',              # the fast, cheap one — great for high-volume
    contents='Say hello in one short sentence.',
)
print(resp.text)                           # the model's reply, as a string
```

genai.Client() makes a client object. generate_content sends the prompt: 'model' picks the variant, 'contents' is your prompt, 'resp.text' is the reply as a string.

gemini-2.5-flash is the fast, cheap variant — the right pick when you tag thousands of reviews.

15

Structured output — describe the shape

For our analyzer, a paragraph reply is a menace

We need label, score and theme as REAL fields, not prose to pick apart. First describe the shape with a Pydantic model (the library that turns a class into a validated data shape), then hand it to Gemini as the required output schema.

```
class Analysis(BaseModel):
    label: str      # positive | negative |
neutral
    score: float    # 0.0 - 1.0
    theme: str      # delivery, price, service,
...
```

```
resp = client.models.generate_content(
    model='gemini-2.5-flash',
    contents=f'Analyze this review:
{review_text}',
    config=types.GenerateContentConfig(
        response_mime_type='application/json',
        response_schema=Analysis,
    ),
)
```

Two settings do the work: response_mime_type='application/json' tells Gemini to reply with JSON, not prose; response_schema=Analysis binds that JSON to our model, so Gemini must return exactly the three fields with the right types.

15

The payoff — resp.parsed

A filled-in object, not a string to wrestle

```
analysis = resp.parsed        # a typed Analysis object, NOT a string

analysis.label      # 'negative'
analysis.score      # 0.2   (already a real number)
analysis.theme      # 'food quality'
```

What you DON'T have to do any more:

- No json.loads
- No stripping the json code fences models love to add
- No guessing if score came back as '0.2' (text) or 0.2 (number)

Pydantic already coerced and validated every field. If Gemini ever returned something malformed, the parse fails LOUDLY — instead of feeding garbage downstream.

15

The pattern beats the vendor

Every LLM SDK is the same three moves

1. Make a client

genai.Client() — same as OpenAI / Anthropic: a client that reads your key

2. Send a message

client.models.generate_content(...) — vs chat.completions.create or messages.create

3. Bind a schema

response_schema=Analysis — vs response_format or a tool definition

Only the attribute names differ. Once you can spot these three moves, you can read ANY LLM SDK cold — even one you've never touched. The vendor is a detail; the pattern is the skill.

15

The back end — api.py

One endpoint; Pydantic validates in and describes out

```
class Review(BaseModel):          # the request body the caller sends
    text: str

@app.post('/analyze', response_model=Analysis)
def analyze(review: Review) -> Analysis:
    resp = client.models.generate_content(
        model='gemini-2.5-flash',
        contents=('Classify the review. label is positive, '
                  'negative, or neutral. score is 0.0 to 1.0. '
                  f'theme is a short noun phrase. Review:
{review.text}'),
        config=types.GenerateContentConfig(
            response_mime_type='application/json',
            response_schema=Analysis),
    )
    return resp.parsed
```

Post without a 'text' field and FastAPI returns a clear 422 error automatically — before our code runs.

One class, two jobs: Analysis is the LLM output schema AND the API's response_model. The contract can never drift apart.

```
Run:  fastapi dev api.py
Then open
127.0.0.1:8000/docs  for a
test page.
```

15

The front end — app.py

A plain script that re-runs top to bottom on every click

```
raw = st.text_area('Paste reviews, one per line:')

if st.button('Analyze'):
    reviews = [l.strip() for l in raw.splitlines() if l.strip()]
    results = []
    for review in reviews:
        resp = requests.post('http://127.0.0.1:8000/analyze',
                             json={'text': review})
        data = resp.json()
        data['review'] = review
        results.append(data)
    st.dataframe(results)          # a sortable table
```

It never calls Gemini. For each review it POSTs to OUR back end and collects the replies into a list, then hands the list to st.dataframe. Aggregates (count, average, % positive) are three one-liners shown with st.metric.

JS -> Python: that for-loop of requests.post is exactly a component fetching an API per item — just synchronous. No await, no .then(): requests.post simply blocks until the response lands. Same idea, calmer syntax.

15

Persistence — database.py

SQLite is a database that lives in a single file — no server

```
def save(results: list[dict]) -> None:
    with sqlite3.connect(DB) as conn:
        conn.executemany(
            'INSERT INTO feedback (review, label, score, theme) '
            'VALUES (?, ?, ?, ?)',
            [(r['review'], r['label'], r['score'], r['theme'])
             for r in results],
        )
```

executemany inserts a whole batch in one go. The 'with sqlite3.connect(...)' block commits and closes the connection for you. init_db uses CREATE TABLE IF NOT EXISTS so it is safe on every startup.

Reading and judging: the ? placeholders are parameterised queries — they prevent SQL injection. NEVER build SQL by string-concatenating user text. Spot a concatenated query and flag it.