Backend Flow — E-commerce Chatbot

1. Overview

The backend is a **stateless FastAPI service** responsible for interpreting user messages, translating them into structured queries, and returning conversational responses enriched with product data.

The backend does **not** maintain long-term user sessions. All context is inferred from each incoming message, making the service horizontally scalable.

2. Request Lifecycle

User Message
↓
/chat endpoint
↓
Text normalization
↓
Intent & constraint extraction
↓
Dynamic SQL generation
↓
BigQuery execution
↓
Response formatting
↓
JSON response to frontend

3. Endpoint Definition

GET `/chat`

Query Parameters

- `message` (string): User's natural language input

Response (JSON)

```
{
  "reply": "Here are jackets under $100 in size M.",
  "products": [],
  "quick_replies": []
}
```

4. Intent Detection Flow

The backend uses lightweight rule-based detection for clarity and explainability.

if message contains "compare" or "difference"
  → comparison intent

else if message contains "gift"
  → gift intent

else
  → search intent

5. Constraint Extraction

Each message is parsed for structured constraints.

Price Parsing

- `under $100` → price ≤ 100
- `over $50` → price ≥ 50
- `$25 jackets` → price ≈ 25 (± tolerance)

Gender → Department Mapping

Phrase	Department
girlfriend, wife, mother	Women
boyfriend, father, man	Men

Size Extraction

User Input	Parsed Size
small	S
medium	M
large	L
xl	XL
xxl	XXL

Category Detection

Matches keywords inside product name:

- jacket
- coat
- hoodie
- shirt
- dress
- pants

6. Dynamic SQL Construction

A base query is progressively enriched with constraints.

```
SELECT name, category, brand, retail_price
FROM products
WHERE 1=1
```

Constraints are appended conditionally:

- Price

- Department
- Size (via name match)
- Category keyword

This approach avoids complex branching logic and keeps queries composable.

7. BigQuery Execution

- Queries are executed using parameterized SQL
- Prevents SQL injection
- Leverages BigQuery's scalability
- Results are limited for UX responsiveness

8. Result Handling

Case A: Results Found

- Products are returned
- Friendly natural-language reply is generated

Example:

> "Here are jackets for women in size M under $100."

Case B: No Results Found

- Bot explains why no products matched
- Suggests recovery actions

Example:


```
{
  "reply": "Sorry, I couldn't find any jackets in size S under $4.",
```

```
  "quick_replies": [
    "Increase budget",
    "Remove size filter",
    "Show similar items"
  ]
}
```

9. Comparison Flow

User: "What's the difference between these two jackets?"
↓
Backend requests product names
↓
Frontend sends selected products
↓
Backend returns comparison data

Comparison is intentionally **explicit** to avoid ambiguity.

10. Error Handling Strategy

- Empty result sets handled gracefully
- Invalid queries default to broad search
- Backend never returns raw errors to UI
- Health check endpoint exposed

11. Scalability Considerations

- Stateless design enables horizontal scaling
- Can be fronted by a load balancer
- Redis can be added for:
    - Context memory
    - Query caching

- LLM can replace rule-based parsing incrementally

12. Why This Design

- Explainable logic (important for demos & interviews)
- Predictable behavior
- Easy to debug
- Easy to extend with ML / LLM later