Chatbot Runtime - How It Works

The accompanying `pizza_chatbot.ipynb` demonstrates the key implementation concepts of a chatbot runtime through a pizza ordering self-service bot. This report details the theory and design decisions behind the implementation.

Core Concepts

Layers of Abstraction

In every chatbot, we have multiple layers of abstraction:

- User Input: The raw text from the user, unprocessed.
- Extracted Actions: What needs to be done based on what we understand from the input.
- Business Logic: External integration and rules based on the specific values we get and domain we operate in.

The goal of a chatbot runtime is to separate these layers cleanly, so that the conversational framework (input processing, dialogue flow) is independent of the business logic (domain-specific rules, data handling) and can be configured. In our example implementation, we achieve this by defining intents and patterns in a JSON file, that the processing pipeline can use independent of the specific domain.

There are two kinds of information that we can win from user input: **intents** and **entities**. The first utterance a user makes usually expresses *what* they want to do and our goal is to link this to a supported intent in our system.

Once we know the intent, we ask follow-up questions, to figure out *how* and *with which values* to perform the necessary actions. To this end, we need to extract **entities**, i.e. valued parameters that are relevant to the intent.

For example, in a pizza ordering bot, the `order` intent needs entities like `pizza_type`, `size`, and `customer_name` to complete the order.

Pizza Bot Domain

The specific intent set for our pizza bot is as follows:

Intent	What it does
`greeting`	Welcome the user
`menu`	Show available pizzas
`order`	Take order + capture name
`order_status`	Look up order (by name if needed)
`goodbye`	End session (only via `exit`/`quit`)
`fallback`	Handle unrecognized input

Our goal here, to achieve the mentioned separation of concerns, is to implement a generic processing pipeline that can handle arbitrary intent sets, and to keep the business logic (e.g. pricing rules) separate from the conversational framework. This is achieved by storing all aspects of an intent (patterns, responses, pricing) in the JSON file, and implementing the business layer (e.g. calculating price based on size and type) in the context management class.

A basic structure could look like this:

```
{
  "intents": [{
    "tag": "order",
    "patterns": ["\\bI'd like\\b", ...],
    "responses": ["Order confirmed! Your order will
     arrive in around 20min."],
    "prices": {"pepperoni": 12},
    "size_modifiers": {"large": 4}
  }]
}
```

This will handle the simple case of a user saying "I'd like a large pepperoni", and the bot responding in a generic way. However, here we have not defined which fields need to be extracted and how the chatbot behaves once we are *within* the `order` Self-Service. To this end, we will also need to model the dialogue flow. This can be an arbitrarily complex graph of states and transitions handling many possible paths. For simplicity, here we will focus on a linear flow, where the bot asks for missing information in a fixed order.

Our goal is to extract the entities `pizza_type`, `size`, and `customer_name`. We can model how the dialogue should behave to collect these entities by simply defining this as part of the intent definition in JSON. Right after finding out that the user wants to place an order, we can try to extract the `pizza_type` and `size` from the initial utterance. If any of these are missing, we can ask for them in turn, by playing an intent confirmation prompt (e.g. "Which pizza would you like?"). Once we have all required entities, we can confirm the order by asking for the `customer_name`, and finally respond with a confirmation message including all relevant details. We can make the strings dynamic by simply including the patterns for that in the JSON:

```
{
  "intents": [{
    "tag": "order",
    ...,
    "confirm_item_prompt": [
        "What type of pizza would you like?
         We have Margherita, Pepperoni, Hawaiian, Veggie Supreme,
          Meat Lovers, and BBQ Chicken.",
        "Which pizza would you like? Options: Margherita,
         Pepperoni, Hawaiian, Veggie Supreme,
          Meat Lovers, or BBQ Chicken."
      ],
      "extract": {
        "item": ["margherita", "pepperoni",
         "hawaiian", "veggie supreme", "veggie",
          "meat lovers", "meat", "bbq chicken", "bbq"],
        "size": ["small", "medium", "large"]
      },
    "confirm_name_prompt": [
        "What name should we put the order under?",
        "Can I get a name for the order?"
      ]
  }]
}
```

Note that in our specific implementation, we use regex in order to match the intent and string matching for entity extraction of item and size. Often, in chatbot frameworks, there are more complex types that the system defines (such as IBAN, date, email, etc.) that can be used for entity extraction. Here, we have a fixed implementation of the name extraction in `handle_name_input`.

The Processing Pipeline

For preprocessing, we apply lowercasing, contraction expansion, and stemming to normalize the input text. This helps with matching user input to patterns defined in the intents. They are applied in sequence before pattern matching (and can be configured via the `CONFIG` dict). The goal is to be able to write very general patterns for regex matching later, that can cover many variations of user input, in spelling and grammar.

Our example processing pipeline looks like this:

Lowercasing, Contraction Expansion, Stemming, Stopword Removal

```
I'd Like to Order a LARGE Pepperoni Pizza
 → i'd like to order a large pepperoni pizza
 → i would like to order a large pepperoni pizza
 → i be like to order a large pepperoni pizza
 → order large pepperoni pizza
```

Pattern matching

We use basic Regex to extract intents. Multiple patterns can be defined per intent, if one of them matches, the intent is selected. They are very useful to capture variations that are not generic enough to be handled by the preprocessing. Some useful regex constructs that are used in our patterns are:

- Word Boundaries: `\b(hi|hello)\b` prevents partial matches inside words
- Optional Elements: `(the)?` allows flexible phrasing
- Alternation: `(get|have|order)` captures multiple ways to express the same thing

Conversation Context Management

To handle multi-turn dialogues we will remember in a chat session what information we have already collected. On the one hand, this information can be used to fill in missing entities for an intent, but it is also relevant to use string templates in the responses (e.g. "Thanks, {name}! Your order will arrive in 20min.").

---

2. Dialogue Tests

Test Case 1: Complete Order Flow

Turn	User Input	Intent	Bot Response	State
1	"Hello!"	greeting	"Hello! Welcome to Pizza Palace!"	-
2	"What's on the menu?"	menu	*[Shows pizza list]*	awaiting=item,size
3	"I'd like a large pepperoni"	order	"What name for the order?"	awaiting=name
4	"John"	(capture)	"Order confirmed, John! Large Pepperoni $16.00"	name=John
5	"Where is my order?"	order_status	"In the oven, John! 15-20 min."	-
6	"exit"	goodbye	"Thanks John!"	ended

Test Case 2: Order Lookup by Name

Someone calling to check on an existing order:

Turn	User Input	Intent	Bot Response	State
1	"Where is my order?"	order_status	"What name is it under?"	awaiting=name
2	"John Smith"	(capture)	"Found it! About 10-15 min."	name=John Smith

Test Case 3: Fallback Recovery

Turn	User Input	Intent	Bot Response
1	"What's the weather?"	fallback	"Sorry, I only handle pizza orders."
2	"I want to order"	order	"Which pizza would you like?"

---

3. Code Overview

```
chatbot-demo/
|-- pizza_chatbot.ipynb    # Implementation
|-- intents.json           # Intent definitions
|-- report.pdf          # This document
```

CONFIG (in notebook)

Runtime settings are contained in in a `CONFIG` dict in the beginning of the notebook together with various default / error prompts. These parameters can be changed to experiment with how well the patterns work when leaving out / enabling certain preprocessing steps.

Setting	What it does
`default_size`	When size not specified
`case_insensitive`	Ignore case in matching
`preprocessing.*`	Toggle normalization steps

---

4. Limitations & Extensions

Current Limitations

- Fixed intent set (need JSON updates to add)
- Single order tracking (no history)
- No quantities ("2 pizzas")
- English only
- No typo handling

Improvements

- Fuzzy matching for misspellings, can order by edit distance to intent patterns
- Multi-item orders, requires NER of count and item pairs
- Delay simulation for backend latency, one can play prompts to the user while asynchronous processing happens
- use spaCy/CRF for entity extraction - handles typos and synonyms
- SVM/BERT instead of regex for better generalization
- use LLMs for personalized responses, apply few-shot prompting to tune the response style and content
- use a vector DB for RAG to get dynamic FAQ, moving away from static actions to combining stored knowledge via an LLM