Pizza Palace Chatbot - Project Report

1. Application Context and Use-Case

Domain: Pizza Ordering Self-Service Bot

The Pizza Palace Chatbot is an interactive conversational agent designed to assist customers with pizza ordering through natural language interaction. The chatbot runs in a Jupyter Notebook environment and provides a complete self-service ordering experience.

Intended Use-Case

The chatbot handles the following customer interactions:

Service	Description
Greeting	Welcome customers and initiate conversation
Menu Inquiry	Display available pizzas and sizes with prices
Order Placement	Take orders with pizza type, size selection, and customer name capture
Order Status	Provide estimated preparation time for active orders, or lookup by name
Exit	End the conversation gracefully (only via `exit` or `quit` commands)

Target Users

- Customers ordering pizza through a text-based interface
- Self-service kiosk applications
- Online ordering chat widgets

---

2. Test Cases

Test Case 1: Complete Order Flow

This test demonstrates a full ordering interaction from greeting to order confirmation.

Turn	User Input	Matched Intent	Bot Response	Context State
1	"Hello!"	greeting	"Hello! Welcome to Pizza Palace! How can I help you today?"	name=None, order=None

Turn	User Input	Matched Intent	Bot Response	Context State
2	"What's on the menu?"	menu	*[Displays pizza menu with sizes and prices]*	name=None, order=None
3	"I'd like a large pepperoni pizza"	order	"Great choice! What name should I put the order under?"	awaiting=name, pending=Pepperoni
4	"John"	(name capture)	"Order confirmed, John! One Large Pepperoni pizza for $16.00. It'll be ready in 15-20 minutes!"	name=John, order=Pepperoni, price=$16.00
5	"Where is my order?"	order_status	"Hi John! Your Large Pepperoni pizza is in the oven. About 15-20 minutes to go!"	name=John, order=Pepperoni
6	"exit"	goodbye	"Thank you for your order, John! Enjoy your pizza!"	Session ended

Key Observations: - Customer name is captured and retained throughout the session - Price is calculated correctly: Pepperoni ($12) + Large (+$4) = $16 - Template substitution works correctly with `{name}`, `{item}`, `{size}`, `{price}`

Test Case 2: Order Status Lookup by Name

This test shows how the chatbot handles status checks when no order has been placed in the current session - it asks for the customer's name to look up their order.

Turn	User Input	Matched Intent	Bot Response	Context State
1	“Good morning!”	greeting	“Greetings! I’m your Pizza Palace assistant. Ready to order a delicious pizza?”	name=None, order=None
2	“Where is my order?”	order_status	“I can look up your order. What name is it under?”	awaiting=name_for_status
3	“John Smith”	(name capture)	“Found it, John Smith! Your order is being prepared. Should be ready in about 10-15 minutes!”	name=John Smith
4	“exit”	goodbye	“Thanks for choosing Pizza Palace, John Smith! See you next time!”	Session ended

Key Observations: - Bot asks for name to look up existing orders (simulates database lookup) - Name is extracted and stored for personalized goodbye - Supports customers calling to check on previously placed orders

---

Test Case 3: Multi-Turn Interaction with Fallback

This test demonstrates handling of unrecognized inputs and conversation recovery.

Turn	User Input	Matched Intent	Bot Response	Context State
1	“Greetings!”	greeting	“Hey! Thanks for choosing Pizza Palace! What can I get for you today?”	greeted=True
2	“What’s the weather like?”	fallback	“Sorry, I only handle pizza orders. Would you like to see the menu or place an order?”	unchanged
3	“I want to order”	order	“What type of pizza would you like? We have Margherita, Pepperoni, Hawaiian, Veggie Supreme, Meat Lovers, and BBQ Chicken.”	awaiting=item
4	“pepperoni”	order	“Perfect! May I have your name please?”	awaiting=name, pending=Pepperoni
5	“Alex”	(name capture)	“Perfect! Medium Pepperoni pizza for Alex, total $14.00. We’re firing up the oven!”	name=Alex, order=Pepperoni
6	“order status”	order_status	“Good news, Alex! Your Medium Pepperoni pizza is being made fresh. Expect it in 15-20 minutes.”	unchanged

Turn	User Input	Matched Intent	Bot Response	Context State
7	“exit”	goodbye	“Thanks Alex! Enjoy your meal!”	Session ended

Key Observations: - Fallback intent gracefully handles off-topic questions - Bot guides user back to supported functionality - Partial order flow (no item specified) is handled by prompting for pizza type - Default size (Medium) applied when not specified

---

3. Data and Code Organization

File Structure

```
chatbot-demo/
|-- pizza_chatbot.ipynb    # Main Jupyter notebook with chatbot implementation
|-- intents.json           # Intent definitions, patterns, and responses
|-- report.md              # This report (source)
+-- report.pdf             # This report (PDF)
```

JSON Schema (intents.json)

The JSON file contains **only intent definitions** - all configuration is in the notebook:

```
{
  "intents": [
    {
      "tag": "intent_name",
      "patterns": ["regex_pattern_1", "regex_pattern_2"],
      "responses": ["response_template_1", "response_template_2"],
      "responses_no_order": [...],      // order_status only (prompts for name)
      "name_prompt_status": [...],      // order_status only
      "responses_lookup": [...],        // order_status only (after name lookup)
      "responses_no_name": [...],       // goodbye only
      "extract": { "item": [...], "size": [...] },  // order only
      "name_prompt": [...],             // order only
      "confirm_item_prompt": [...],     // order only
      "prices": { "pepperoni": 12 },    // order only
      "size_modifiers": { "small": 0 }  // order only
    }
  ]
}
```

Each intent contains only the fields it needs - no redundant null values or unused configuration.

Notebook Configuration (CONFIG dict)

All runtime settings are centralized in a `CONFIG` dictionary in the notebook:

Setting	Description
`default_size`	Default pizza size when not specified
`case_insensitive`	Case-insensitive pattern matching
`max_name_words`	Max words to extract from name
`preprocessing.*`	Text normalization options (see below)
`empty_input_message`	Response for empty input
`error_message`	Response for exceptions
`name_only_response`	Response when name given without order
`order_error_message`	Response when order fails
`fallback_goodbye`	Goodbye when intent not found
`interrupt_goodbye`	Goodbye on keyboard interrupt
`no_match_response`	Response when no intent matches

Preprocessing options: `expand_contractions`, `lowercase`, `remove_punctuation`, `stem`, `remove_stopwords`

Code Organization (Notebook Sections)

Section	Purpose
1. Dependencies	Install and import NLTK and standard libraries
2. Import Libraries	Load re, json, random, typing, nltk modules
3. Configuration & Load	`CONFIG` dict and `load_intents()` function
4. Text Preprocessing	Contraction expansion, tokenization, stemming
5. Pattern Matching	`match_intent()` and `extract_order_details()` functions
6. Context Management	`ConversationContext` class for state tracking
7. Response Generation	`generate_response()` and intent-specific handlers
8. Main Loop	`run_chatbot()` interactive conversation loop

Section	Purpose
9-12. Testing	Automated test cases and pattern verification

Key Functions

Function	Description
`load_intents()`	Load and parse JSON intents
`preprocess_text()`	Normalize text using CONFIG settings
`match_intent()`	Match input against regex patterns
`extract_order_details()`	Extract pizza type and size from order
`generate_response()`	Generate context-aware response
`handle_name_for_status()`	Process name for order status lookup

---

4. Technical Strategy

4.1 Text Preprocessing

The preprocessing pipeline normalizes user input for robust pattern matching. All steps are configurable via `CONFIG['preprocessing']`:

1. **Contraction Expansion**: Converts contractions to full forms
   - "I'd" → "I would"
   - "What's" → "What is"
   - "Can't" → "Cannot"
2. **Lowercasing**: Ensures case-insensitive matching
3. **Punctuation Removal**: Strips punctuation that could interfere with patterns
4. **Tokenization**: Splits text into individual words
5. **Stemming**: Reduces words to root forms using Porter Stemmer
   - "ordering" → "order"
   - "pizzas" → "pizza"

4.2 Pattern Matching

Regular expressions are used for intent recognition with the following techniques:

Technique	Example	Purpose
Word boundaries	`\b(hi|hello)\b`	Prevent partial matches
Case insensitivity	`re.IGNORECASE`	Match any capitalization (configurable)
Optional elements	`(the)?`	Handle optional words
Character classes	`\s`, `\w`	Match whitespace/word characters
Quantifiers	`*`, `+`, `?`	Variable-length patterns
Alternation	`(get|have|order)`	Match multiple phrasings

Example Pattern (Order Intent):

```
\b(i('?d|\s*would)?\s*like|i\s*want)\b
```

This pattern matches "I'd like", "I would like", "I want" with word boundaries.

4.3 Context Handling and Business Logic Injection

The `ConversationContext` class serves a dual purpose:

1. **State Management**: Tracks conversation state across turns
2. **Business Logic Injection Point**: Domain-specific operations are implemented here

```
class ConversationContext:
    customer_name: str          # Customer's name for personalization
    current_order: dict         # {item, size, price}
    awaiting: str               # Current expected input type
    pending_item: str           # Item waiting for name confirmation
    has_active_order: bool      # Whether an order exists

    def calculate_price(self, intent):  # <-- BUSINESS LOGIC
        # Domain-specific pricing rules injected here
```

Architectural Separation:

Layer	Responsibility	Example
Conversational Framework	Intent matching, dialog flow, response generation	`match_intent()`, `generate_response()`
Business Logic Layer	Domain-specific operations, calculations, validation	`calculate_price()`, `set_order()`

Layer	Responsibility	Example
Configuration	Runtime settings, messages	`CONFIG` dict in notebook
Intent Data	Patterns, responses, prices	`intents.json`

This separation is critical: the conversational framework remains **domain-agnostic**, while all domain-specific logic (pricing, validation, inventory checks) is injected through the context class. To adapt this chatbot to a different domain (e.g., hotel booking), one would:

1. Create a new `intents.json` with hotel-specific intents
2. Implement a new context class with hotel business logic (room pricing, availability checks)
3. Reuse the same conversational framework unchanged

State Transitions: - `awaiting='item'` → User must specify pizza type - `awaiting='name'` → User must provide their name (for order) - `awaiting='name_for_status'` → User must provide their name (for order lookup) - `awaiting=None` → Normal intent matching

4.4 Response Generation

Responses are generated through:

1. **Random Selection**: Multiple response templates per intent for variety
2. **Template Substitution**: Placeholders replaced with context values
   - `{name}` → Customer name
   - `{item}` → Pizza type
   - `{size}` → Size selection
   - `{price}` → Calculated total

---

5. Implementation Decisions and Limitations

Design Decisions

Decision	Rationale
Clean JSON schema	Intent definitions only - no configuration or defaults in JSON
Centralized CONFIG	All runtime settings in one place, easy to modify
No hardcoded strings	All user-facing messages configurable via CONFIG
Regex over ML	Simpler, deterministic, sufficient for limited domain

Decision	Rationale
Porter Stemmer	Lightweight, no training required, good for English
Context class	Clean state management, business logic injection point
Random response selection	Provides variety, feels more natural

Limitations

1. **Fixed Intent Set**: Adding new intents requires JSON updates
2. **Single Order Tracking**: Only tracks the most recent order
3. **No Quantity Support**: Cannot order "2 pepperoni pizzas"
4. **English Only**: Patterns and preprocessing tuned for English
5. **No Spelling Correction**: Misspelled pizza types won't match

Potential Improvements

- Add fuzzy matching for misspellings
- Support multiple items per order
- Implement order history with database backend
- Add configurable delay simulation to mimic backend latency

Future Extensions

The current regex-based approach could be extended with:

- **Classical ML (NER/Slot Filling)**: Replace regex extraction with trained models (CRF, spaCy NER) for more robust entity recognition—handling typos, synonyms, and unseen phrasings.
- **Intent Classification**: Use supervised classifiers (SVM, BERT) instead of pattern matching for better generalization across varied user inputs.
- **Generative AI**: Integrate LLMs to generate personalized responses, handle open-ended FAQs, or provide conversational fallbacks when intents don't match.