🤖Task 3: Smart Customer Support Chatbot — End-to-End Starter Kit

This kit gives you a production-ready path to design, build, and deploy a customer support chatbot on **web (Streamlit)** and **Telegram**, with **FAQ retrieval**, **intent/entity recognition**, and **smart fallback & human handoff**. It also shows how to adapt the same design in **Dialogflow** or **Rasa**.

---

1) Architecture at a Glance

- **UI**: Streamlit web app + optional Telegram bot.
- **Core Brain**: OpenAI GPT (tools: retrieval, ticketing, escalation) + lightweight intent/entity layer.
- **Knowledge**: FAQs/Docs (CSV/Markdown) → embeddings → similar question search.
- **Fallbacks**: clarifying questions → guided forms → human handoff (email/Jira/Zendesk stub).
- **Analytics**: conversation logs, intent stats, fallback rate, CSAT thumbs.

```
User → (Streamlit/Telegram) → Router (intent/entity + retrieval) → GPT with
context → Response
↘️                                          (low confidence) → Fallback/Handoff
```

---

2) Project Structure

```
support-bot/
├── data/
│   ├── faq.csv                # id, question, answer, tags
│   └── intents.csv            # id, intent_name, sample_utterance
├── src/
│   ├── app_streamlit.py       # Web UI
│   ├── bot_telegram.py        # Telegram webhook/poller
│   ├── router.py              # intent/entity detection + retrieval + policy
│   ├── retrieval.py           # embedding index build/query
│   ├── ticketing.py           # handoff stubs (email/Jira/Zendesk placeholder)
│   ├── eval_intents.py        # quick intent evaluation script
│   └── utils.py
├── models/
│   └── index.pkl              # saved vector index + metadata
├── .env                       # OPENAI_API_KEY, TELEGRAM_BOT_TOKEN, etc.
├── requirements.txt
└── README.md
```

`requirements.txt`

```
python-dotenv>=1.0
pandas>=2.0
numpy>=1.24
openai>=1.0.0
scikit-learn>=1.3
streamlit>=1.31
python-telegram-bot>=20.0
fastapi>=0.110
uvicorn>=0.29
pydantic>=2.0
```

> If you later use Rasa/Dialogflow, add their CLIs separately. For vector DB, you can start with scikit-learn KNN (included) and later switch to FAISS/PGVector.

---

3) Data Formats (Starter)

`data/faq.csv`

```
id,question,answer,tags
1,How can I reset my password?,Use the Reset Password link on the login page. If you don't receive the email, check spam or contact support.,account
2,How to track my order?,Go to Orders > Track. Enter your order ID to see real-time status.,orders,tracking
3,What are your refund timelines?,Refunds are processed within 5–7 business days after approval.,billing,refunds
```

`data/intents.csv`

```
id,intent_name,sample_utterance
1,reset_password,forgot password
2,order_status,track my order
3,refund_policy,refund timeline
4,human_handoff,talk to a human
```

---

4) Retrieval (Embeddings + KNN)

Create `src/retrieval.py`:

```python
import pickle
import pandas as pd
import numpy as np
from sklearn.neighbors import NearestNeighbors
from openai import OpenAI

client = OpenAI()

EMBED_MODEL = "text-embedding-3-small"

class FAQIndex:
    def __init__(self, k=5):
        self.k = k
        self.knn = None
        self.matrix = None
        self.df = None

    def build(self, faq_path: str):
        self.df = pd.read_csv(faq_path)
        texts = self.df['question'].astype(str).tolist()
        vecs = self._embed(texts)
        self.matrix = np.array(vecs)
        self.knn = NearestNeighbors(n_neighbors=min(self.k, len(texts)),
metric='cosine')
        self.knn.fit(self.matrix)
        return self

    def save(self, path: str):
        with open(path, 'wb') as f:
            pickle.dump({
                'matrix': self.matrix,
                'df': self.df
            }, f)

    def load(self, path: str):
        with open(path, 'rb') as f:
            obj = pickle.load(f)
            self.matrix = obj['matrix']
            self.df = obj['df']
            self.knn = NearestNeighbors(n_neighbors=min(self.k, len(self.df)),
metric='cosine')
            self.knn.fit(self.matrix)
        return self

    def query(self, text: str, topk=3):
        vec = np.array(self._embed([text]))
        dists, idxs = self.knn.kneighbors(vec, n_neighbors=min(topk,
```

```
len(self.df)))
        results = []
        for d, i in zip(dists[0], idxs[0]):
            row = self.df.iloc[i]
            results.append({
                'id': int(row['id']),
                'question': str(row['question']),
                'answer': str(row['answer']),
                'distance': float(d)
            })
        return results

    def _embed(self, texts):
        resp = client.embeddings.create(model=EMBED_MODEL, input=texts)
        return [d.embedding for d in resp.data]
```

Build once at startup (see `router.py`).

---

5) Router (Intent/Entity + Policy + Fallback)

Create `src/router.py`:

```
import os
import re
import pandas as pd
from dataclasses import dataclass
from typing import Dict, Any
from openai import OpenAI
from .retrieval import FAQIndex

client = OpenAI()
CHAT_MODEL = os.getenv('CHAT_MODEL', 'gpt-4o-mini')

@dataclass
class RouteResult:
    reply: str
    intent: str
    confidence: float
    citations: list
    handoff: bool = False

class Router:
    def __init__(self, faq_path: str, intents_path: str):
        self.faq = FAQIndex().build(faq_path)
```

```
        self.intents = pd.read_csv(intents_path)

    # very light entity examples (extend as needed)
    def extract_entities(self, text: str) -> Dict[str, Any]:
        order_id = re.findall(r"\b\d{6,}\b", text)
        email = re.findall(r"[\w\.-]+@[\w\.-]+", text)
        return {"order_id": order_id[0] if order_id else None, "email": email[0]
if email else None}

    def detect_intent(self, text: str) -> Dict[str, Any]:
        # Use GPT to classify intent names from intents.csv
        intent_list = ", ".join(sorted(self.intents['intent_name'].unique()))
        sys = (
            "You are an intent classifier. Return JSON with fields: intent,
confidence (0-1). "
            f"Valid intents: {intent_list}. If unknown, use 'unknown'."
        )
        msg = [
            {"role":"system","content":sys},
            {"role":"user","content":text}
        ]
        r = client.chat.completions.create(model=CHAT_MODEL, messages=msg,
temperature=0)
        # naive parse
        content = r.choices[0].message.content
        intent = 'unknown'
        conf = 0.5
        try:
            import json
            obj = json.loads(content)
            intent = obj.get('intent','unknown')
            conf = float(obj.get('confidence',0.5))
        except Exception:
            pass
        return {"intent": intent, "confidence": conf}

    def generate(self, user_text: str, history: list):
        ents = self.extract_entities(user_text)
        ic = self.detect_intent(user_text)

        # Retrieve FAQs
        hits = self.faq.query(user_text, topk=3)
        retrieved_context = "\n\n".join([f"Q: {h['question']}\nA: {h['answer']}"
for h in hits])
        citations = [h['id'] for h in hits]

        # Policy
        if ic['intent'] == 'human_handoff' or ic['confidence'] < 0.35:
```

```
            # smart fallback with clarifying questions
            reply = (
                "I can connect you to a human agent. Before that, could you
share a few details "
                "(short description, email, and any order ID)?"
            )
            return RouteResult(reply=reply, intent=ic['intent'],
confidence=ic['confidence'], citations=citations, handoff=True)

        # Compose prompt
        sys = (
            "You are a concise, helpful support assistant. Use the provided FAQ
context first. "
            "If an answer is not in context, say you don't have that info and
suggest talking to an agent. "
            "Keep answers under 120 words unless asked for more."
        )
        msgs = [
            {"role":"system","content":sys},
            {"role":"user","content":f"User: {user_text}\n\nContext:
\n{retrieved_context}"}
        ]
        r = client.chat.completions.create(model=CHAT_MODEL, messages=msgs,
temperature=0.2)
        answer = r.choices[0].message.content
        return RouteResult(reply=answer, intent=ic['intent'],
confidence=ic['confidence'], citations=citations)
```

---

6) Streamlit App (Web)

Create `src/app_streamlit.py`:

```
import os
import streamlit as st
from dotenv import load_dotenv
from datetime import datetime
from router import Router

load_dotenv()

st.set_page_config(page_title="Support Bot", page_icon="🟫")
st.title("🟫 Smart Support Chatbot")
st.caption("Real-time support with FAQ retrieval, intents & smart fallback")
```

```
FAQ = os.getenv('FAQ_PATH', 'data/faq.csv')
INTENTS = os.getenv('INTENTS_PATH', 'data/intents.csv')
router = Router(FAQ, INTENTS)

if 'history' not in st.session_state:
    st.session_state.history = []   # list of {role, content}

user_text = st.chat_input(placeholder="Ask me about orders, refunds, account…")

for msg in st.session_state.history:
    with st.chat_message(msg['role']):
        st.markdown(msg['content'])

if user_text:
    st.session_state.history.append({"role":"user","content":user_text})
    with st.chat_message("user"):
        st.markdown(user_text)

    outcome = router.generate(user_text, st.session_state.history)

    with st.chat_message("assistant"):
        st.markdown(outcome.reply)
        if outcome.citations:
            st.caption(f"Sources: {', '.join(map(str, outcome.citations))}")
        if outcome.handoff:
            with st.expander("Contact a human agent"):
                email = st.text_input("Your email")
                desc = st.text_area("Brief description")
                if st.button("Request Callback"):
                    st.success("Thanks! An agent will reach out soon.")


st.session_state.history.append({"role":"assistant","content":outcome.reply})
```

Run:

```
streamlit run src/app_streamlit.py
```

---

7) Telegram Bot

Create `src/bot_telegram.py`:

```python
import os
from dotenv import load_dotenv
from telegram import Update
from telegram.ext import ApplicationBuilder, ContextTypes, MessageHandler,
CommandHandler, filters
from router import Router

load_dotenv()
TOKEN = os.getenv('TELEGRAM_BOT_TOKEN')
FAQ = os.getenv('FAQ_PATH', 'data/faq.csv')
INTENTS = os.getenv('INTENTS_PATH', 'data/intents.csv')
router = Router(FAQ, INTENTS)

async def start(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text("Hi! I'm your support bot. Ask me anything
about orders, refunds, or accounts.")

async def echo(update: Update, context: ContextTypes.DEFAULT_TYPE):
    text = update.message.text
    res = router.generate(text, [])
    await update.message.reply_text(res.reply)

def main():
    app = ApplicationBuilder().token(TOKEN).build()
    app.add_handler(CommandHandler("start", start))
    app.add_handler(MessageHandler(filters.TEXT & ~filters.COMMAND, echo))
    app.run_polling()

if __name__ == "__main__":
    main()
```

For webhooks, deploy with FastAPI/uvicorn and set Telegram webhook URL; polling is OK for demos.

---

8) Environment & Secrets

Create `.env`:

```
OPENAI_API_KEY=sk-...
CHAT_MODEL=gpt-4o-mini
FAQ_PATH=data/faq.csv
INTENTS_PATH=data/intents.csv
TELEGRAM_BOT_TOKEN=123456:ABC-DEF...
```

Never commit real keys.

---

9) Smart Fallback & Handoff (Patterns)

1. **Low confidence** (`intent_conf < 0.35`): ask a clarifying question; offer agent.
2. **Missing entities** (e.g., no order ID): ask for it; show example.
3. **Policy-guard**: if user asks for unsupported actions (refund approval), explain limitation + handoff.
4. **Timeout / multi-turn**: after 2 unclear turns, escalate.

`src/ticketing.py` (stub):

```
import smtplib
from email.message import EmailMessage

def send_email_ticket(to, subject, body):
    # Fill with your SMTP details or helpdesk API
    msg = EmailMessage()
    msg['Subject'] = subject
    msg['To'] = to
    msg['From'] = 'no-reply@company.com'
    msg.set_content(body)
    # smtp = smtplib.SMTP_SSL('smtp.gmail.com', 465)
    # smtp.login('user','pass')
    # smtp.send_message(msg)
    return True
```

---

10) Dialogflow CX Alternative (Blueprint)

- **Agents & Flows**: One main flow with intents: `reset_password`, `order_status`, `refund_policy`, `human_handoff`.
- **Entities**: `sys.number` (order_id), `sys.email`.
- **Routes**: NLU route to FAQ webhook that performs retrieval; default route for fallbacks (max 2) → handoff.
- **Webhook**: Cloud Function/Cloud Run endpoint implementing the same retrieval logic from `retrieval.py`.
- **Parameters & Forms**: For order status, collect `order_id` then call backend.
- **Rich responses**: suggestion chips for common follow-ups.

---

11) Rasa Alternative (Starter Files)

`domain.yml` (excerpt):

```
intents:
  - reset_password
  - order_status
  - refund_policy
  - human_handoff
entities:
  - order_id
slots:
  order_id:
    type: text
    influence_conversation: true
responses:
  utter_ask_order_id:
    - text: "Please share your order ID (6+ digits)."
  utter_handoff:
    - text: "Connecting you to a human agent. One moment."
```

`data/nlu.yml` (excerpt):

```
nlu:
- intent: reset_password
  examples: |
    - forgot password
    - reset my password
- intent: order_status
  examples: |
    - where is my order
    - track my order 123456
- intent: refund_policy
  examples: |
    - refund timeline
    - how long do refunds take
- intent: human_handoff
  examples: |
    - talk to a human
    - connect me to an agent
```

`data/stories.yml` (excerpt):

```
stories:
- story: Order status flow
  steps:
  - intent: order_status
```

```
  - slot_was_set:
    - order_id: null
  - action: utter_ask_order_id
  - intent: order_status
  - action: action_check_order
```

Custom action calls your retrieval/backends.

---

12) Evaluation & QA

- **Intent accuracy**: per-class precision/recall; confusion matrix (use `eval_intents.py`).
- **Answer quality**: manual rubric + CSAT thumbs in UI.
- **Fallback rate**: target < 15% after tuning.
- **Latency**: p95 < 2.5s (enable response streaming if needed).
- **Safety**: block PII leaks; refuse policy-violating requests.

`src/eval_intents.py` (mini):

```python
import pandas as pd
from sklearn.metrics import classification_report
# provide a test file with columns: text,true_intent,pred_intent
# then print classification_report
```

---

13) Deployment

- **Web (Streamlit Cloud / Render / HuggingFace Spaces)**: push repo, set env vars, run `streamlit run src/app_streamlit.py`.
- **Telegram**: host `bot_telegram.py` on a small VM; for webhooks, add FastAPI endpoint and set webhook via BotFather.
- **Observability**: log chats (sans secrets), export weekly metrics (fallback %, top intents, avg CSAT).

---

14) Submission Checklist

- [ ] Working web chatbot URL (Streamlit).
- [ ] Optional Telegram bot handle.
- [ ] `faq.csv`, `intents.csv`, and code repo with `src/`.
- [ ] README with setup, env vars, and screenshots.
- [ ] Short report: architecture diagram, metrics, and recommendations.

---

15) Tips

- Start with clear FAQs; 30–60 high-quality entries beat 300 noisy ones.
- Add guided forms for flows that need entities (order ID, email, product).
- Use retrieval first, then model creativity; bound answers to company policy.
- Iterate using logs: add training phrases where users fail.

Drop your FAQ and a few real conversation snippets; I'll tailor the intents/entities and plug your domain terms directly into the router and prompts.