

Large Language Models for Customer Support

Daniel Valle Alvarez

University of Split

Index

Introduction: Current AI solutions for Customer Service3

 Zendesk.....3

 Help Scout.....4

Hardware5

Training5

 Models5

 Database7

 Training code.....8

Usage..... 12

 API..... 13

 Front end22

Application and conclusion25

Bibliography27

Introduction: Current AI solutions for Customer Service

This section will act as a brief report about information found online about companies that provide customer support based on AI. As a new sector, it is still largely uncharted territory and it is blooming right now, so there are a lot of new solutions, most of them without a large customer base. Some of the ones that have gotten most traction will be described here.

Zendesk

The Company was founded in 2007 in Copenhagen, raised about \$86 million dollars in venture capital, went public in 2014 and was acquired by investors in 2022 for approximately \$10.2 billion. [1,2]

At its core, Zendesk serves as a centralized hub for managing customer interactions across multiple channels. The platform enables businesses to collect and organize customer inquiries from email, chat, phone, social media, and messaging apps. It tracks customer issues through customizable ticketing systems, provides self-service options through knowledge bases and AI-powered help centers, and generates insights on support performance and customer satisfaction metrics.[3]

Zendesk's AI capabilities are primarily delivered through its Answer Bot technology, which is built on natural language processing (NLP) and machine learning algorithms. This technology enables automated responses, as the Answer Bot can analyze incoming customer queries and automatically suggest or provide relevant responses from the company's knowledge base. The AI identifies the customer's intent from their message, which helps route inquiries to the appropriate departments or suggest suitable solutions. Additionally, Zendesk's AI can detect customer sentiment, allowing support teams to prioritize urgent or negative interactions, and analyze conversation patterns to identify trends in customer issues. It now also offers integration with Copilot to improve its chat capabilities, can also include advanced statistics and activity tracking for the human agents even acting as a QA.[3,4]

The price of the service depends on the services offered, for standard companies and small businesses it ranges between \$20 and \$115 per agent per month in its annual plan. Copilot, QA and the tools to track employee's performance are paid as an add-on. It can work with email systems, like a live chat or and provides tools for streamlining phone communications.[5]

Help Scout

This company was founded in 2011 in Massachusetts. Unlike some competitors that heavily emphasize AI as their primary value proposition, Help Scout takes a more balanced approach that prioritizes human-centered support enhanced by technology.[6,7]

The core architecture of Help Scout revolves around its shared inbox system, which serves as the foundation for its support operations. This system uses sophisticated routing algorithms to distribute incoming customer communications across support teams based on predefined rules including expertise matching, workload balancing, and availability status. The platform implements a collaborative editing framework allowing multiple agents to work on responses simultaneously while preventing conflicting edits through real-time synchronization protocols.[7,8,9]

From a technical perspective, Help Scout's knowledge base functionality employs semantic search technology to improve the accuracy of self-service support. The search engine utilizes natural language processing to understand query intent rather than simply matching keywords. This is achieved through contextual analysis that considers word relationships, synonyms, and common support terminology. The system continuously refines its search algorithms by analyzing successful query resolutions and incorporating this feedback into future search results.[7,8,9]

Help Scout integrates machine learning capabilities primarily focused on workflow optimization rather than customer-facing automation. Its prediction algorithms analyze historical support data to identify patterns in issue resolution, using this information to suggest relevant knowledge base articles to agents during customer interactions. It can also detect customer frustration, prioritizing those tickets.[7,8,9,10]

Its pricing, for free for up to 50 unique clients a month but also offer two other "standard" paid plans and an option to contact their sales team in order to implement a more tailored system. Of course, the more expensive plans include more features. Its service is more focused on emails and, as mentioned previously, it is oriented towards providing faster and more accurate responses from a human team by making access to relevant information easier and faster and organizing the inboxes.[7,9,11]

Hardware

As it might be of relevance, I will include in this section the hardware used for this project.

Asus ROG Zephyrus G16 GU605MV laptop with the following relevant components:

Intel Core Ultra 9 185H with 6 performance cores, 8 efficiency cores and 2 low power efficiency cores. It includes an incorporated GPU, with Intel Arc Graphics.

32 Gb of DDR5 at 7467MT/s

NVIDIA GeForce RTX 4060 Laptop GPU

Training

Seeing my very limited options to use part of the commercial solutions as a baseline for my project, I had to create the customer support agent from the beginning.

I tried and tested several models, trying to find a compromise between time spent training and results obtained through its usage.

Models

T5-Small

The T5 Small model is part of Google's Text-To-Text Transfer Transformer (T5) family, a series of models designed to unify various natural language processing (NLP) tasks under a single framework. T5 reformulates every problem as a text-to-text task, meaning inputs and outputs are always treated as strings of text, making it different from other models that use different architectures for tasks like translation, summarization, or classification. This approach simplifies fine-tuning and allows the model to generalize across a wide range of applications without requiring task-specific modifications.

T5 Small was trained using the Colossal Clean Crawled Corpus (C4), a dataset of scraped and cleaned web text, utilizing the same methodology as its larger counterparts. The training process uses masked language modeling (MLM) and span corruption, where random sections of text are masked and the model learns to predict the missing content. Both techniques are similar, the main difference being that span corruption does not only mask one token, it can also mask a sequence of them. Other techniques used were translation and judging the grammatical acceptability of a sentence.

One of T5 Small's biggest advantages is its efficiency. It can run on lower end hardware, including consumer-grade GPUs or even CPUs with reasonable speeds. Tuning the model for a new task is straightforward, thanks to its architecture, as users only need to format their data in an input-output text pair structure (such as the one we have).

However, its smaller size also means reduced reasoning depth compared to larger models. It may struggle with highly complex queries or tasks requiring deep contextual awareness. It can also struggle with tasks that require common sense or real-world experience. We must understand that the Text-to-Text is oriented towards creating input-output pairs in a more deterministic way, so it can struggle in more "creative" tasks. Nevertheless, it excels in simple clear input output, as may be translating a text. As our use case is a chatbot, it is well suited to perform this task.[12,13]

Phi-2

Phi-2 is a 2.7 billion parameter language model developed by Microsoft Research. It is designed for tasks like reasoning, language understanding, and coding while maintaining a smaller size compared to other modern AI models.

The model was trained on 1.4 trillion tokens. The training data mixture contains synthetic datasets specifically created to teach the model common sense reasoning and general knowledge. The training corpus was augmented with web data that is filtered based on educational value and content quality.

Phi-2 has a strong focus on its efficiency. Even though it is heavier than T5-Small, it can run on consumer-grade hardware while still delivering strong performance in different tasks.

However, Phi-2 has some limitations. It performs best in standard English and may struggle with informal language or multilingual tasks, but in general, as we see an increase in the amount of parameters, the languages will be more precise when answering to our queries for customer service and they will be better at understanding different contexts. [14,15]

Phi-3-mini

Phi-3 Mini is a 3.8 billion-parameter language model from Microsoft's Phi-3 family, and, as the previous models, with a focus on efficiency and performance in resource-constrained environments. It is designed to handle tasks like reasoning, coding, and language understanding while being small enough to run locally on small devices.

The model was trained on 3.3 trillion tokens of data, including filtered web content and synthetic datasets tailored for reasoning and educational purposes. As in the case of Phi-2, there was a heavy focus on educational content and high-quality information. Post-training, and unlike the previously described models, it underwent a post-training process that involved supervised fine-tuning.

As mentioned previously, there is a clear improvement on Phi-3-Mini over Phi-2 on the processing capabilities, especially on those referring to understanding the context of different prompts, but there is still space to improve in comparison to much larger models like Chat-GPT 3 or similar. [16,17]

Database

It is designed to fine-tune Large Language Models (LLMs, like the ones used) for customer support automation. The purpose of this database is to be used as a baseline for training the LLM and later do a second tuning with a second smaller database, from customer interactions from the company. As that level of specificity is not needed, that second step will not be taken, with the intent of leaving us with a more generalist model.

The dataset is hybrid. That means that originally it used natural texts as source, with real customer service dialogues (emails, chats). Later an NLP Extraction was performed, where key phrases ("seeds") were pulled from these texts. Other LLMs were then used to create variations based on those previous seeds, and were added to the database. Finally, there were computational linguists who reviewed the process and the entries, to ensure that they were accurate and natural.

It organizes the instruction-response pairs into categories, quite general and that could be used in almost any business:

ACCOUNT: Create, delete, edit, or switch accounts.

CANCELLATION_FEE: Check fees for canceling services.

DELIVERY: Ask about shipping options.

FEEDBACK: Handle complaints or reviews.

INVOICE: Retrieve or check invoices.


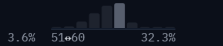



NEWSLETTER: Manage subscriptions.

ORDER: Cancel, modify, or place orders.

PAYMENT: Resolve payment issues.

REFUND: Check policies or track refunds.

SHIPPING_ADDRESS: Update delivery details.

flags	instruction	category	intent	response
string · classes	string · lengths	string · classes	string · classes	string · lengths
				
BLQZ	3.6% 51*60 32.3%	ORDER 14.8%	cancel_ord_ 3.7%	1.27k*1.51k 2.1%
BKZ	canceling order {{Order Number}}	ORDER	cancel_order	I realized you're looking for assistance with canceling your order with the order number_
BCLQW	i purchased some fucking item cancel order {{Order Number}}	ORDER	cancel_order	I'm aware that you're frustrated with your purchase and would like to cancel order {{Order_
BLQZ	want assistance to cancel pyrchase {{Order Number}}	ORDER	cancel_order	I'm with you, your need for assistance in canceling purchase {{Order Number}}. Your_
BQ	i need assistance to cancel order {{Order Number}}	ORDER	cancel_order	I've noticed that you need assistance with canceling order {{Order Number}}, and I'm here t_
BQ	i need help with canceling order {{Order Number}}	ORDER	cancel_order	I acknowledge that you need assistance with canceling your purchase of order number {{Order_
BK	cancellng order {{Order Number}}	ORDER	cancel_order	I grasp your request to cancel order {{Order Number}}. Our main priority is to assist you in_
BCELV	I have ordered some item, cancel purchase {{Order Number}}	ORDER	cancel_order	I'm conscious of the reality that you need assistance with canceling your purchase with the_

BELPQZ	i do not know hoow i could check ur reimbursement policy	REFUND	check_refund_policy	I'm cognizant of the fact that you're unsure about how to access our reimbursement policy. N_
BLZ	help me check your money bck guarantee	REFUND	check_refund_policy	Sure! I understand your need to check our money-back guarantee. Our company takes pride in_
BLQ	need assistance to check in what cases can i request refunds	REFUND	check_refund_policy	I hear you loud and clear your request for assistance in understanding the cases in which_

Some flags are also added to the orders, as to reflect how language varies/changes across different linguistic phenomena like colloquial or offensive language. These tags indicate the type of language variation that the entry expresses. They help train the model to adapt tone based on context.

In summary, the dataset contains an extensive amount of text data across its 'instruction' and 'response' columns, which after tokenizing, amount to a total of 3.57 million tokens. This provides very broad coverage to all businesses models that are trained with this model and a cost-effective mechanism to train them, as the companies don't need massive proprietary datasets, just a small second fine-tuning step. [18]

Training code

I have written different code for each one of the different models, but all are quite similar as the techniques for training them are almost the same. I will use the code for Phi-3-Mini as a reference. In general, unless the parameters for the several functions that we will use are of special interest, I will refrain from explaining them, as it will make the explanation tremendously long.


```
# 1. Login & GPU check
token = open("token.txt").read().strip()
login(token=token)
print(f"CUDA: {torch.cuda.is_available()}, GPU: {torch.cuda.get_device_name(0)}")
```

After the initial imports, the next section takes care of login in into Hugging Face (necessary for accessing some models and databases in this platform). In order not to have to write the token in the code (that is uploaded in GitHub) or having to enter it manually each time I tried to run the program, I just wrote it into the first line of a file available locally in the main folder of my computer.

The last line of this section is just a check. As I was having trouble with the different libraries and running the code on my laptop's dedicated GPU, it checks that torch has cuda available (NVIDIA GPU) and prints the device name. In my case a lot of the problems stem from training this AI on a laptop. I had almost all the libraries installed for a previous project (training XGBoost on my GPU), but I had problems (in both cases) that I think were derived from the MUX switch that the computer uses to avoid using the discrete GPU for lighter tasks. Once it was deactivated, and only the dGPU used, problems with it disappeared. Still this step was a useful check to know if the program was behaving as expected.

```
# 2. Load dataset
dataset = load_dataset("bitext/Bitext-customer-support-llm-chatbot-training-dataset")

def formatting_func(sample):
    text = f"<|user|>\n{sample['instruction']}\n<|assistant|>\n{sample['response']}<|end|>"
    return text
```

Next step is loading the dataset, which is done in the first instruction. Afterwards we define a formatting function that will be used in the trainer. This function has slight variations in between models, sometimes it includes truncation. In this cases it simply appends user, assistant and end to distinguish between roles and to mark the end of the conversation.

```

# 3. Load Phi-3-mini with QLoRA (no Flash Attention)
model_id = "microsoft/Phi-3-mini-4k-instruct"
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True, # QLoRA
    bnb_4bit_compute_dtype=torch.float16, # Windows-friendly
)

tokenizer = AutoTokenizer.from_pretrained(model_id, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    device_map="auto",
    torch_dtype=torch.float16, # Windows-compatible
)
model = prepare_model_for_kbit_training(model)

```

In this section we first define the model (Phi-3-Mini) and later we define the parameters for QLoRA. QLoRA or Quantized Low-Rank Adaptation is a technique that I used in the fine tuning of the model in order to reduce memory usage. It uses two different mechanisms to enable lower resource utilization. The first one is quantization, compressing the pre-trained model's weights from 32-bit or 16-bit precision to 4-bit, to reduce memory usage. The other is mentioned in its name (LoRA), low rank adaptation. It introduces small, trainable matrices (adapters) into the model layers instead of updating all parameters. After training it merges them into the model, preserving the original architecture.

Afterwards we set up the tokenizer, necessary to convert the input into tokens, numerical values that represent words, subwords, or characters that represent chunks of raw text.

Finally, we load the model with the previously defined parameters (including the QLoRA quantization parameters that I will not delve into) and we “freeze” the models weights in the last line, in order to use QLoRA training and not modify the original weights.

```
# 4. LoRA Config (optimized for 8GB)
peft_config = LoraConfig(
    r=8, # Reduce to 4 if OOM
    lora_alpha=16,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
model = get_peft_model(model, peft_config)
```

This next screenshot we can see the hyperparameters for QLoRA. The most important is rank (first one), where that defines the rank of the matrices for LoRA. A lower rank reduces memory usage and increases speed during training but can lead to underfitting (model behaving worse in complex situations).

```
# 5. Trainer Config (Windows-safe)
sft_config = SFTConfig(
    output_dir="./results/phi3-qlora-results",
    num_train_epochs=3,
    per_device_train_batch_size=1, # Conservative for Windows
    gradient_accumulation_steps=4,
    learning_rate=3e-4,
    optim="paged_adamw_8bit", # Helps with memory spikes
    max_seq_length=512, # Reduced for Windows stability
    fp16=True, # Enabled for Windows
    bf16=False, # Disabled (Windows issues)
    gradient_checkpointing=True, # Critical for 8GB
    logging_steps=10,
    save_steps=500,
)

trainer = SFTTrainer(
    model=model,
    args=sft_config,
    train_dataset=dataset["train"],
    formatting_func=formatting_func,
)
```

In this screenshot we see the different parameters used for the trainer. Most of the settings are related to reducing the usage of resources (we are quite constrained, especially memory wise).

```
# 6. Train
trainer.train()
model.save_pretrained("./results/phi3-customer-support-qlora")
print("Training complete!")
```

Finally, we train and save the model. The length of the process varies significantly depending on which model we train. The shortest one was T5-Small at around two hours, while Phi-3-Mini took around 20 hours.

Other considerations

As you can see if you check the repository, they were not the only models that were planned to be trained. Originally, the first model I tried was Mistral 7B, but the waiting times for it were so big that I ended up dropping it (around 50 or 60 hours). I could have train it if the code run in my home computer (it has a way more powerful GPU), but both, the Python modules, and the drivers drivers and extra configuration from NVIDIA refused to work. They can run on my laptop because I had already done some AI projects for another subject here in Croatia. I tried to get it running for a couple of hours, but in the end, I thought it was not worth it, this project is more of a “proof of concept” than a “technical demo” of GPUs or LLMs. It is the same reason why I did not contact you asking if you had an extra GPU or if you could run the training for me. Also, it made debugging a lot easier, because as the models were trained on my computer, I was sure they could fit on my computer’s VRAM while I ran a lot of tests ensuring the responses and their formatting was correct, so it helped narrowing down possible errors.

Usage

The application I ended up building is composed of two parts. The first one is an API that is in charge of communication with the LLM models, sending the user queries and delivering their response to the front end. The other is the front end itself, a React app through which the user chooses the model, sends messages and visualizes the response.

API

The code in the API is written in Python. FastAPI, creates a web server that allows us to interact with the models, by defining the endpoints for the models and handling HTTP requests and responses. CORS allows us to receive requests from the front end (browsers would block requests from the front-end).

I ended up only supporting two models, as I was totally unable to make T5-Small work. I tried for a long time but there were problems somewhere, and I could not receive a single cohesive answer from it. I don't know if it is related to the size of tokens and messages as it gave me errors in the training, and I had to truncate some messages, as they were exceeding the maximum length (I am not sure if this approach was correct, but is what I saw online, and went with it as it did not gave me any more problems). Anyhow there were no errors registered in the logs, no warnings even and the lack of informations in and aorund the answers diffculted the debugging.

I will now try to explain the code, as previously, I will not delve into specific parameters (most of them were not even optimized, they just run), as I feel that is outside the scope of the project.

```
15 # Fix for Windows logging encoding
16 sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')
17 sys.stderr = io.TextIOWrapper(sys.stderr.buffer, encoding='utf-8')
18
19 # Create offload directory if it doesn't exist
20 os.makedirs("./offload", exist_ok=True)
```

After the initial imports, the first lines ensure correct formatting in the logs, enforcing utf-8 coding.

We later create a folder (called offload) if it does not exist. At some point, Phi-3 required it for offloading as we were memory constrained. Offloading means moving parts of the model from the GPU VRAM or the CPU's RAM to secondary storage. It can make inference slower, but at least the model will run.

```

22 # Configure logging with UTF-8 encoding
23 class UTF8StreamHandler(logging.StreamHandler):
24     def __init__(self, stream=None):
25         super().__init__(stream)
26
27     def emit(self, record):
28         try:
29             msg = self.format(record)
30             stream = self.stream
31             stream.write(msg + self.terminator)
32             self.flush()
33         except Exception:
34             self.handleError(record)
35
36 dictConfig({
37     "version": 1,
38     "formatters": {
39         "default": {
40             "format": "[%(asctime)s] %(levelname)s in %(module)s: %(message)s",
41         }
42     },
43     "handlers": {
44         "console": {
45             "()": UTF8StreamHandler,
46             "stream": sys.stdout,
47             "formatter": "default",
48         },
49         "file": {
50             "class": "logging.FileHandler",
51             "filename": "api_errors.log",
52             "formatter": "default",
53             "encoding": "utf-8",
54         }
55     },
56     "root": {
57         "level": "DEBUG",
58         "handlers": ["console", "file"]
59     }
60 })
61

```

In this next section, we configure logging. I won't go deep into it, as it is not that related to the project.

```

62     app = FastAPI()
63
64     # CORS setup
65     app.add_middleware(
66         CORSMiddleware,
67         allow_origins=["*"],
68         allow_methods=["*"],
69         allow_headers=["*"],
70     )
71

```

Here we create the instance for the app, and we configure CORS. In this case we allow requests from any domain, with any methods (POST, GET...) and with any headers. I know this is not safe for a finished app, but I don't think this app will go into production any time soon.

```

72 # Model management
73 class ChatRequest(BaseModel):
74     user_input: str
75     model_name: str = "default"
76     context: Optional[str] = None
77
78 # Combined model configurations - Phi-2 from first code, Phi-3 from second code
79 MODEL_CONFIGS = {
80     "phi2-support": {
81         "path": "./models/phi2",
82         "type": "causal",
83         "tokenizer": "microsoft/phi-2",
84         "prompt_template": "<|system|>\n{context}\n<|user|>\n{input}\n<|assistant|>\n",
85         "response_processor": lambda response: response.split("<|assistant|>\n")[-1].split("<|end|>")[0].strip()
86     },
87     "phi3-support": {
88         "path": "./models/phi3",
89         "type": "causal",
90         "tokenizer": "microsoft/phi-3-mini-4k-instruct",
91         "prompt_template": (
92             "<|system|>\n{context}<|end|>\n"
93             "<|user|>\n{input}<|end|>\n"
94             "<|assistant|>\n"
95         ),
96         "response_processor": lambda response: response.split("<|assistant|>")[-1].split("<|end|>")[0].strip()
97     }
98 }

```

Here we define the structure of the API requests. We receive an input, the model we want to send that input to and optionally a context.

Later we see the two models and their configuration in a dictionary. The causal type specifies that the type of the model, in this case causal or autoregressive. Causal is commonly used for chat generation, it means it predicts the next token in a sequence based only on the preceding tokens. It cannot “see” future tokens, it

works left to right. A lambda function extracts the response from the assistant's reply from the raw output.

```
100 # Track the currently loaded model
101 current_model_name = None
102 loaded_components = None
103
104 def cleanup_model():
105     """Clean up the currently loaded model and free memory."""
106     global loaded_components, current_model_name
107
108     if loaded_components is not None:
109         model, tokenizer, _ = loaded_components
110         try:
111             # Move model to CPU first to ensure proper cleanup
112             if hasattr(model, 'to'):
113                 model.to('cpu')
114
115             # Delete model and tokenizer
116             del model
117             del tokenizer
118
119             # Clear CUDA cache
120             if torch.cuda.is_available():
121                 torch.cuda.empty_cache()
122
123             # Force garbage collection
124             gc.collect()
125
126             logging.info(f"Successfully unloaded model: {current_model_name}")
127         except Exception as e:
128             logging.error(f"Error unloading model {current_model_name}: {str(e)}", exc_info=True)
129         finally:
130             loaded_components = None
131             current_model_name = None
132
```

In this section we first track what model is loaded. The `cleanup_model` function is used to unload the model from the memory (necessary as the user can change in between them and we do not have memory enough to keep both loaded). It first tries to move the model to the CPU and deletes the reference to the model and the tokenizer. Then forces the GPU to clean the cache and runs the garbage collector to ensure the recovery of the memory.


```

133 def load_model_and_tokenizer(model_name: str):
134     global loaded_components, current_model_name
135
136     try:
137         if model_name not in MODEL_CONFIGS:
138             raise ValueError(f"Unknown model: {model_name}")
139
140         # If requesting a different model than currently loaded, cleanup first
141         if current_model_name is not None and current_model_name != model_name:
142             cleanup_model()
143
144         # Load new model if not already loaded
145         if loaded_components is None:
146             logging.info(f"Loading model: {model_name}")
147             config = MODEL_CONFIGS[model_name]
148
149             # Load tokenizer
150             tokenizer = AutoTokenizer.from_pretrained(config["tokenizer"])
151             tokenizer.pad_token = tokenizer.eos_token
152
153             # Model-specific loading configurations
154             if model_name == "phi2-support":
155                 # Phi-2 loading from first code (working configuration)
156                 model = AutoModelForCausalLM.from_pretrained(
157                     config["path"],
158                     device_map="auto",
159                     torch_dtype=torch.float16,
160                 )
161             elif model_name == "phi3-support":
162                 # Phi-3 loading from second code (working configuration)
163                 quantization_config = BitsAndBytesConfig(
164                     load_in_4bit=True,
165                     bnb_4bit_compute_dtype=torch.float16,
166                     bnb_4bit_quant_type="nf4",
167                 )
168
169                 model = AutoModelForCausalLM.from_pretrained(
170                     config["path"],
171                     device_map="auto",
172                     torch_dtype=torch.float16,
173                     trust_remote_code=True,
174                     offload_folder="./offload",
175                     quantization_config=quantization_config,
176                     attn_implementation="eager"
177                 )
178
179             loaded_components = (model, tokenizer, config)
180             current_model_name = model_name
181
182     return loaded_components

```

This function is in charge of loading the different models. It raises an exception if the model is not found in the dictionary. It later cleans the model if there is a change. The next section loads the tokenizer and defines the parameters for the different models. Phi2 loads with standard parameters, but we use quantization in Phi-3 to reduce memory use. It is not seen in the screenshot because it did not fit, but if the try fails it raises an exception.

```

@app.get("/models")
async def get_available_models():
    return {"models": list(MODEL_CONFIGS.keys())}

@app.post("/chat")
async def chat(chat_request: ChatRequest):
    try:
        logging.info(f"Chat request received for model: {chat_request.model_name}")
        model, tokenizer, config = load_model_and_tokenizer(chat_request.model_name)

        # Build prompt according to model-specific template
        prompt = config["prompt_template"].format(
            context=chat_request.context or "No context provided",
            input=chat_request.user_input
        )

        logging.debug(f"Generated prompt: {prompt}")

        # Model-specific tokenization
        if chat_request.model_name == "phi2-support":
            # Phi-2 tokenization from first code
            inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512).to("cuda")
        else: # phi3-support
            # Phi-3 tokenization from second code
            max_length = 4096
            inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=max_length).to("cuda")

```

In this next section we define the endpoints that the front end will use to interact with the API. The get for the models returns a list with the name of the available models in the API.

Later we are seeing part of the implementation of the post “/chat” used for sending information to the model (we will continue in the next screenshots). First it loads the model and necessary tokenizer through the previously explained functions. It later builds the prompt for the tokenizer, using the template found in the dictionary at the beginning of the code. It specifies the context or the lack of it and adds the user input.

```

205 # Model-specific tokenization
206 if chat_request.model_name == "phi2-support":
207     # Phi-2 tokenization from first code
208     inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512).to("cuda")
209 else: # phi3-support
210     # Phi-3 tokenization from second code
211     max_length = 4096
212     inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=max_length).to("cuda")
213
214 # Model-specific generation configurations
215 if chat_request.model_name == "phi2-support":
216     # Phi-2 generation config from first code (working)
217     generation_config = {
218         "max_new_tokens": 150,
219         "temperature": 0.5,
220         "do_sample": True,
221         "top_p": 0.9,
222         "repetition_penalty": 1.5,
223         "num_beams": 4,
224         "early_stopping": True,
225         "no_repeat_ngram_size": 3
226     }
227 else: # phi3-support
228     # Phi-3 generation config from second code (working)
229     generation_config = {
230         "max_new_tokens": 512,
231         "temperature": 0.7,
232         "top_p": 0.9,
233         "repetition_penalty": 1.1,
234         "no_repeat_ngram_size": 3,
235         "pad_token_id": tokenizer.eos_token_id,
236         "eos_token_id": tokenizer.eos_token_id,
237         "do_sample": True,
238         "use_cache": False
239     }
240
241 outputs = model.generate(
242     **inputs,
243     **generation_config
244 )
245

```

In this screenshot we see the continuation of the previous endpoint. This next step is tokenization, with small differences in between the language, as Phi-3 admits longer prompts. If the maximum length of the prompt is surpassed, it is truncated to fit. We specify that we want it to be formatted in “cuda” as it is running on an NVIDIA GPU.

This made me realize the code is probably not portable to many other computers, as big parts of it are dependent on the GPU brand, and even some on the generation, as some libraries are specific for certain GPUs. If this was going to work on corporate servers, we would probably need to adapt it to run on whatever hardware they were running or use some cloud computing solution.

Anyhow, coming back to the code, next we have the specific parameters for the generation of the response. `max_new_tokens` ensures that the response is not too long. `temperature` is a parameter that defines the randomness of the response, how much is the model allowed to “diverge” from the learnt responses. `do_sample` is another way of introducing randomness. Instead of just using a greedy algorithm to choose the next most likely token (making the model more repetitive) we use sampling to make it a bit more “human like”. To avoid weird outliers, we limit tokens, only including the ones that can be found in the 90% more probable (`top_p`). `repetition_penalty` is self-explanatory. `num_beams` forces the model to explore 4 parallel sequences at each step, keeping the most likely overall path. It improves

coherence but it is more expensive computationally. When we finalize one of the beams naturally the model stops its generation (early_stopping). The last token just forces the model to not repeat any 3-word sequence.

I will go one step further and try to explain how they work in combination (the ones that do) as I feel it is important to understand how do LLMs work. First, we have a series of previous tokens in the response. To use the same example I have seen, let's say they were "The sky is". The model is trying to predict the next token, so it pulls a probability list of them, and they are: blue (0.6), gray (0.2), clear (0.15), banana (0.05). We then apply the temperature. Temperature increases the difference between the probabilities if it is below one and decreases it if it is above 1. In this case, it will increase the difference in probabilities (0.5), changing them to blue (0.72), gray (0.18), clear (0.09) and banana (0.01). We will later select the ones that make up the cumulative 90% (blue and gray), and sample in between them, picking blue in 72 cases out of 90.

To place the beams in this example, each time we continue the sequence of tokens, we will sample four times in each beam, and, out of the 16 (if none have stopped yet) we score them using a heuristic and keep the fourth highest scorers that are not repeated.

As I mentioned previously, these parameters were not optimized at all, they were the ones that were used first, and as they worked, I did not change them. If this was a more "professional" project, done by a corporation, I am sure they would be fine-tuned.

```

246     # Decode the full response
247     full_response = tokenizer.decode(outputs[0], skip_special_tokens=False)
248     logging.debug(f"Full model response: {full_response}")
249
250     # Process response according to model type
251     response = config["response_processor"](full_response)
252
253     # Additional cleanup for Phi-3 (from second code)
254     if chat_request.model_name == "phi3-support":
255         # Remove any remaining special tokens
256         response = response.replace("<|assistant|>", "").replace("<|end|>", "").replace("<|endoftext|>", "").strip()
257         # Remove the prompt if it was included in the response
258         if prompt in response:
259             response = response.replace(prompt, "").strip()
260         # Clean up any remaining special characters
261         response = ''.join(char for char in response if char.isprintable())
262
263     logging.info(f"Processed response (truncated for logs): %s", response[:200])
264
265     return {
266         "response": response,
267         "model_used": chat_request.model_name,
268         "model_type": config["type"]
269     }
270
271 except Exception as e:
272     logging.error(f"Error in chat endpoint: {str(e)}", exc_info=True)
273     raise HTTPException(status_code=500, detail=str(e))
274

```

In this next section we do the post processing. We first decode the message received using the tokenizer (changing the tokens back to normal language). We later use the lambda function in the dictionary to format the response. As Phi-3 was giving me a lot of troubles to solve formatting, a new section was added here, to ensure it was written correctly. In the end, we send the response obtained back to the front end. If anything fails during the process, we raise an exception.

```

275 @app.on_event("shutdown")
276 def shutdown_event():
277     """Clean up models when the application shuts down."""
278     cleanup_model()
279
280 if __name__ == "__main__":
281     import uvicorn
282     uvicorn.run(app, host="0.0.0.0", port=8000)

```

In the last screenshot, the `on_event` ensures that when we stop the API, no models are still loaded in the GPU. The method is deprecated, but it still works.

The last lines tell uvicorn to run the app on port 8000, locally.

Front end

The front end is done in React. I will just explain the code in the app.js, as I don't feel like the styles are necessary to understand how does this app works (.css) and I have not modified the index (in a production environment I would have done it to include the corporate logo and give a name to the tab, for example).

```
4  function App() {
5      const [input, setInput] = useState('');
6      const [context, setContext] = useState('');
7      const [messages, setMessages] = useState([]);
8      const [isLoading, setIsLoading] = useState(false);
9      const [availableModels, setAvailableModels] = useState([]);
10     const [selectedModel, setSelectedModel] = useState('');
11     const [showContextInput, setShowContextInput] = useState(false);
12 }
```

After the imports, we initialize the variables. Their name is self-explanatory.

```
13     // Fetch available models on component mount
14     useEffect(() => {
15         async function fetchModels() {
16             try {
17                 const response = await fetch('http://localhost:8000/models');
18                 const data = await response.json();
19                 setAvailableModels(data.models);
20                 if (data.models.length > 0) {
21                     setSelectedModel(data.models[0]);
22                 }
23             } catch (error) {
24                 console.error('Error fetching models:', error);
25             }
26         }
27         fetchModels();
28     }, []);
29 }
```

We later have the fetchModels() function. It asks the API for the available models and then sends them to another function. It defaults to the first one if none are selected. It manages errors if the API raises them. It is contained in an useEffect() with an empty dependency array, so it runs once only, when the app gets rendered for the first time.

```

30     const handleSubmit = async (e) => {
31         e.preventDefault();
32         if (!input.trim() || !selectedModel) return;
33
34         setIsLoading(true);
35         const userMessage = { text: input, sender: 'user' };
36         setMessages(prev => [...prev, userMessage]);
37
38         try {
39             const response = await fetch('http://localhost:8000/chat', {
40                 method: 'POST',
41                 headers: { 'Content-Type': 'application/json' },
42                 body: JSON.stringify({
43                     user_input: input,
44                     model_name: selectedModel,
45                     context: context.trim() || undefined
46                 })
47             });
48
49             const data = await response.json();
50             setMessages(prev => [...prev, {
51                 text: data.response,
52                 sender: 'bot',
53                 model: data.model_used
54             }]);
55         } catch (error) {
56             console.error('Error:', error);
57             setMessages(prev => [...prev, {
58                 text: "Sorry, something went wrong.",
59                 sender: 'bot',
60                 model: 'error'
61             }]);
62         }
63
64         setInput('');
65         setIsLoading(false);
66     };
67

```

This next function, `handleSubmit` is in charge of sending the user messages. First it checks if the input is not empty and if there is a selected model and then sets the state of the app to loading. It adds the sent message to the Messages list (as it is an `useState`, it forces a rerender).

It later sends the API the chat POST request, with all needed information for the API, and waits for the answer. Once it receives it, it adds it to the list, empties the input and stops the loading state. It catches any errors on the API's end.

```

68   const handleStartNewChat = () => {
69       setMessages([]);
70       setContext('');
71   };
72
73   const handleModelChange = (e) => {
74       setSelectedModel(e.target.value);
75       handleStartNewChat();
76   };
77
78   const handleKeyDown = (e) => {
79       if (e.key === 'Enter' && !e.shiftKey) {
80           e.preventDefault();
81           handleSubmit(e);
82       }
83       // Shift+Enter will work as default (new line)
84   };
85

```

The last functions define the behavior of the app when creating a new chat (empties the messages list and cleans the input text), when it changes the model (creates a new chat) and when enter is pressed (it submits the message unless enter it is pressed with shift, where it just adds a new line)

The last part of the code just contains components of the site. I will not describe them in much detail, as they are generally only aesthetic, other components would also work. Some of the components are disabled during loading, to avoid any unwanted state in the app. It also contains placeholders for when the model is loading and explaining how does the text area work.

Application and conclusion

The finished app looks like this:

Customer Support Chat POC

Model: phi2-support Add Context New Chat

my headphones broke

I'm sorry to hear that your headphones broke. I understand how frustrating this can be, especially when you rely on them for your audio needs. To assist you further, could you please provide some more context about the issue? Specifically, let me know if the headphones stopped working suddenly or if there were any warning signs leading up to the breakage. This information will help us determine the best course of action to resolve the problem and get you back to enjoying your audio experience.

Model: phi2-support

The earpad tore in half, I would need a replacement, they are under warranty

I'm sorry to hear that your earpad has torn in half and you require a replacement. Rest assured, we understand the importance of having a functional product, especially when it's under warranty. To assist you further, could you please provide me with the specific details of your purchase, such as the order number or any other relevant information? This will enable us to locate your order and initiate the replacement process promptly. Thank you for bringing this to our attention, and we appreciate your patience as we work towards resolving this issue for you.

Model: phi2-support

Type your message... (Shift+Enter for new line)

Send

They work as expected. I found phi-3 responses to be better in general (as expected by using a bigger and more complete model). As we did not do the second fine tuning with a database of interactions for our business, the responses tend to be quite open, not concise or exact, but correct, nonetheless. They ask for information that could be useful and they have behaved as anticipated. In general, I found them specially good at “pointing” the customer on the right direction.

The model does not have access to a real business database (I don't know exactly how that could be implemented), so it can't make queries to check data. Because of this I don't know if the model would just take everything the client says at face value or if it will check that information. Also, I can't check how it will behave if the client is lying.

Analyzing the project as a whole, and its possible applications in the real world, the main problem I see is that I don't think any corporation's board would approve of having an LLM model at the front of customer interaction. We have seen how it is possible to "cheat" models way more complex (probably more than any medium-size business could afford to fine tune and run) like chat-GPT, making them say things they are not supposed to. Having a company representative (in this case the model) using foul language could be detrimental to the image of the corporation, but this could be the smaller of the problems they could face. If the model is given access to the customer's database, that generates a massive security gap. If a model like this could be tricked into using colorful language, users might be able to trick it into giving personal information of other customers, leaving the company liable to legal action. More problems may appear if we allow the model to take real decisions about sending replacements or modifying package information, or in general, take real action. If models can be tricked, could they be convinced to enact a reimbursement for a product that has never been purchased?

This leaves us with the most general question possible. What do we use them for? Well, if we give them real responsibilities, that comes with great risk. If the models are fine tuned for long and tested extensively (none of those things were done for this project), the risks might decrease, and a company might take their chances and assume some of the losses as a counterbalance for the savings in personnel or as the expense of a PR campaign. Most companies, however, will probably use them as the big retailers are doing, to try and screen some of the users' problems from the real human agents. This, however, as a user of those services myself, it becomes really frustrating, really fast, as no one wants to talk with an agent, spend their time explaining a problem they cannot solve, only to be redirected to a human agent, to whom you have to explain it again. I have personally seen this trend in some sites, like Amazon (in my case Amazon.es), where they tried to "hide" their human agents behind an AI chat. They ended up having to backtrack, as almost every customer just ended up writing in the chat "I want to talk with a human" and paying their frustration with the company customer support.

In conclusion, Large Language Models are definitely an interesting field to explore in the area of customer support, but the creativity they have and is necessary for other areas, can act in detrimental ways in this scenario, as real unsupervised responsibility could bring dire consequences.

Bibliography

[1] J. Shinal, "Startup strives to make customer service a more 'zen' experience," NBC News, Jul. 31, 2011. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.nbcnews.com/id/wbna43960527>

[2] Zendesk, "Consortium led by Hellman & Friedman and Permira completes acquisition of Zendesk," Zendesk, Nov. 22, 2022. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.zendesk.com/newsroom/press-releases/consortium-led-by-hellman-friedman-and-permira-completes-acquisition-of-zendesk/>

[3] Zendesk and E. O'Callaghan, "Getting started with Zendesk AI and Copilot," Zendesk help. Accessed: Jun. 29, 2025. [Online]. Available: <https://support.zendesk.com/hc/en-us/articles/5608652527386-Getting-started-with-Zendesk-AI-and-Copilot>

[4] Zendesk and P. Couch, "About generative AI features in Zendesk," Zendesk help. Accessed: Jun. 29, 2025. [Online]. Available: <https://support.zendesk.com/hc/en-us/articles/6059285322522-About-generative-AI-features-in-Zendesk>

[5] Zendesk, "Zendesk Pricing," Zendesk. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.zendesk.com/pricing/#compare-plans>

[6] R. Empson, "Help Scout Draws Funding To Bring Affordable Email Collaboration And Support To Startups," TechCrunch, Oct. 08, 2008. Accessed: Jun. 29, 2025. [Online]. Available: <https://techcrunch.com/2025/06/27/only-2-exhibitor-tables-up-for-grabs-at-techcrunch-all-stage-claim-by-june-29/>

[7] N. Timms, "Help Scout Pricing: is it worth it (and are there alternatives)?," Drag. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.dragapp.com/blog/help-scout-pricing/>

[8] Help Scout, "Help Scout Integrations: Connect Help Scout with 100+ Apps," Help Scout. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.helpscout.com/help-desk-integration/>

[9] Help Scout, "User Based Billing and Plans Guide," Help Scout Support. Accessed: Jun. 29, 2025. [Online]. Available: <https://docs.helpscout.com/article/596-billing-and-plans-guide>

[10] Help Scout, "Help Scout," The Shared Inbox with Unlimited Seats. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.helpscout.com/inbox/>

[11] Help Scout, “Help Scout Pricing: Affordable Customer Support Software,” Help Scout. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.helpscout.com/pricing/>

[12] Google, “t5-small ,” Hugging Face. Accessed: Jun. 29, 2025. [Online]. Available: <https://huggingface.co/google-t5/t5-small>

[13] Dataloop, “T5 Small,” Dataloop. Accessed: Jun. 29, 2025. [Online]. Available: https://dataloop.ai/library/model/google-t5_t5-small/

[14] Microsoft, “Phi 2,” Hugging Face. Accessed: Jun. 29, 2025. [Online]. Available: <https://huggingface.co/microsoft/phi-2>

[15] A. Hughes, “Phi-2: The surprising power of small language models,” Microsoft Research. Accessed: Jun. 29, 2025. [Online]. Available: <https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/>

[16] S. Beatty, “Tiny but mighty: The Phi-3 small language models with big potential,” Source. Accessed: Jun. 29, 2025. [Online]. Available: <https://news.microsoft.com/source/features/ai/the-phi-3-small-language-models-with-big-potential/>

[17] Microsoft, “Phi-3-mini 4k instruct,” Hugging Face. Accessed: Jun. 29, 2025. [Online]. Available: <https://huggingface.co/microsoft/Phi-3-mini-4k-instruct>

[18] Bitext, “Bitext-customer-support-llm-chatbot-training-dataset,” Hugging Face. Accessed: Jun. 29, 2025. [Online]. Available: <https://huggingface.co/datasets/bitext/Bitext-customer-support-llm-chatbot-training-dataset>