Platform for Chatbots Powered by Controllable Large Language Models: Backend Development

By Morgan Cupp (mmc274)
Advisors: Qian Yang and Michael Hedderich

Introduction

The purpose of this project was to create a system for building customizable chatbots that leverage large language models, or LLMs for short. Such a system empowers non-technical individuals to benefit from the capabilities of modern LLMs such as OpenAI's GPT-3. This is an important step towards making the technology more accessible and having its benefits be more widespread. Furthermore, the system allows very general LLMs to be easily adapted to more specific use-cases. This system will first be deployed on Social Media TestDrive to create chatbots that teach students about upstanding cyberbullying. I was tasked with, and will be discussing, the backend development for this project.

Relevant Background

Language models work by taking in a string of text and generating a response. The "prompt" is the input to the model, and the responses are generated by predicting one new word at a time. Language models learn how to respond through exposure to human-created examples.

LLMs are built upon the same principles as language models, except at a massive scale. They contain more learnable parameters and more complex neural network architectures, enabling them to make better predictions. The other key ingredient to LLMs is a vast quantity of training data, which is on the scale of all the text on the internet. As a result, LLMs are more general.

Two key concepts employed in this system are prompt chaining and few-shot learning. Prompt chaining refers to decomposing complex prompts into simpler prompts. For example, instead of saying "summarize this paragraph and translate it to French", one might first say "summarize this paragraph", get the summary, and then say "translate this to French". Interacting with LLMs through these simpler, chained prompts has been shown to yield better results.

Few-shot learning refers to the ability of LLMs to learn using a small number of examples embedded in the prompt. These examples guide the model's output by giving it a few correct examples to learn from in the prompt. This allows LLM users to fine-tune the model without needing vast quantities of data.

Implementation Overview: Dialogue Tree

The chatbot designer is based on the concept of dialogue trees. Dialogue trees store and facilitate conversations in role-playing games, basic chatbots, and more. Traditionally, a user is presented with fixed input options, chooses one, and then a hard-coded response is returned. This process repeats until the conversation is over. This project's system aims to be more natural and flexible by utilizing the generative abilities of LLMs.

A similar dialogue tree representation is used to store each custom chatbot. Dialogue trees consist of detection and generation components connected by edges. Detection components classify user inputs, and generation components respond to them.

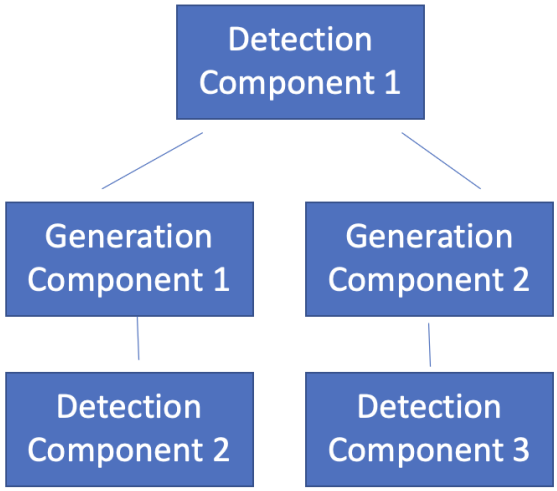

Figure 1: High-level dialogue tree that consists of detection components, generation components, and edges.

Implementation Overview: Components

Detection components require classes and few-shot examples provided by the chatbot designer. Each few-shot example is an example user utterance that would be labeled with a particular class name. At runtime, the classes and few-shot examples are inserted into the prompt.

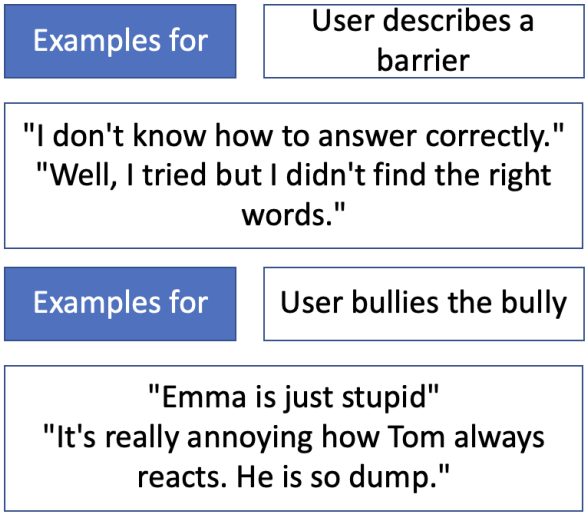

Figure 2: High-level detection component that consists of two classes and two examples per class.

Generation components require one class and few-shot examples provided by the chatbot designer. The class describes the type of user inputs the component will generate responses to. For generation components connected to detection components, the generation class must match one class in the preceding detection component. For chained generation components, the class can be any descriptive name. Each few-shot example consists of a context and response. The context is a sample utterance the component must respond to. For generation components following detection components, this will be the most recent user message. For generation components following other generation components, this will be the most recent chatbot message. The responses are example chatbot responses. At runtime, the class, contexts, and responses are similarly incorporated into the prompt.

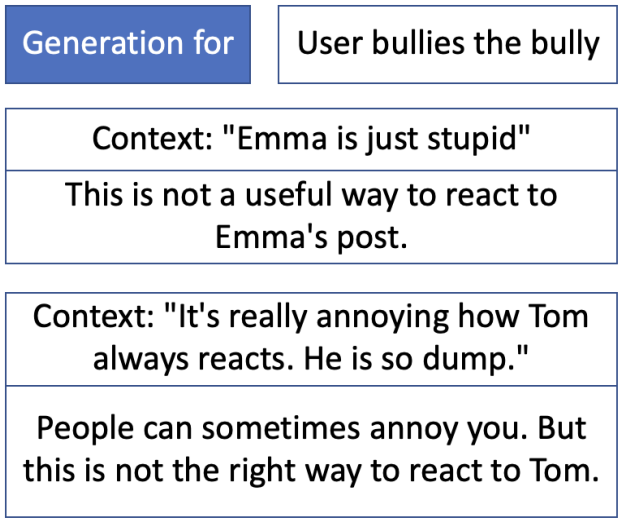

Figure 3: High-level generation component that consists of a class and two examples. Each example contains a context and response.

Implementation Overview: Traversing Dialogue Tree

At runtime, the chatbot designer's dialogue tree facilitates an interactive conversation with the user. The conversation messages are stored as conversation history. Traversing the dialogue tree occurs in the following loop. The user sends a message, which is added to the conversation history. This conversation history is then sent to a detection component. The detection component classifies the most recent message, which is from the user. Based on the detected class, the appropriate generation component is selected. The generation component generates a response to the most recent user message. In the case of chained generation components, the latter generation components respond to the former generation component outputs. All generation component responses are added to the conversation history. Once either a second detection component or tree leaf is reached, the tree traversal stops. In the case of a second detection component, a new user message gets added to the conversation history, and then the loop repeats.

Implementation Details: File Structure and Data Model

The backend code is organized into five Python files. *app.py* contains 30 APIs written using Flask. *models.py* contains the class definitions and methods that define the data model for the dialogue tree. *validation.py* contains functions to validate each API request for correctness prior to processing it. *helpers.py* contains all other miscellaneous helper functions, and should perhaps be refactored in the future. Lastly, the *data* directory locally stores one *.pkl* file per dialogue tree that is read and written via the API.

Dialogue trees are internally represented using an object-oriented tree data structure. The high-level class structure is shown below.

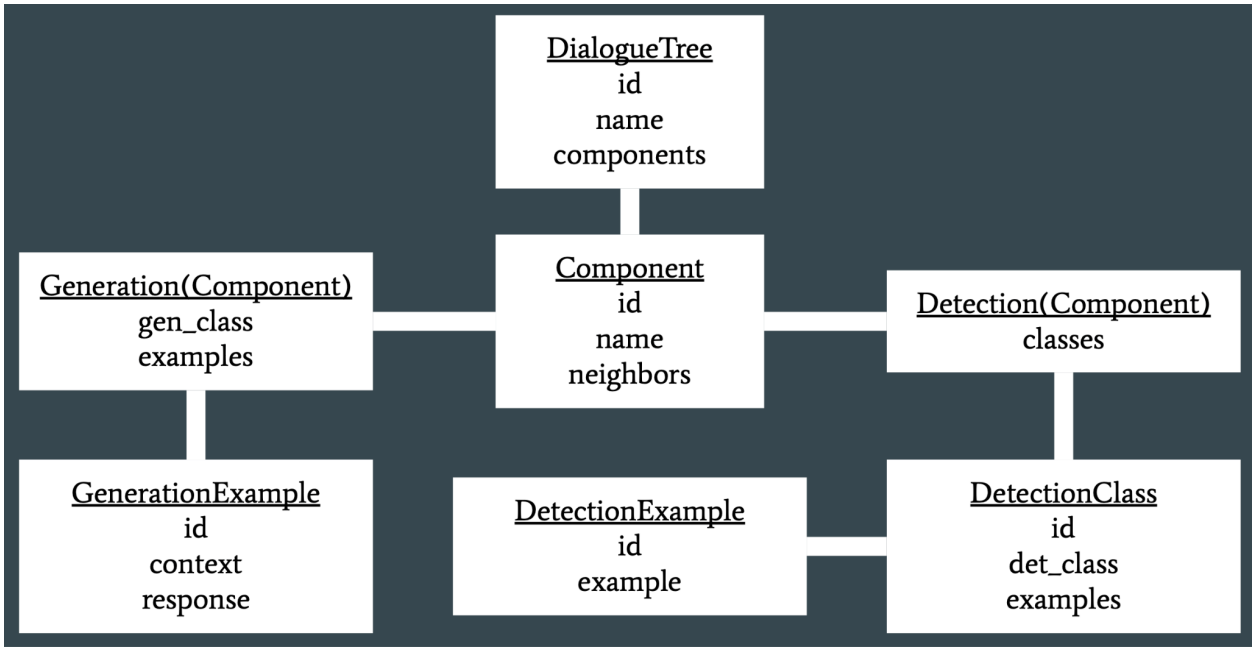

Figure 4: Object-oriented data model for creating, modifying, and storing dialogue trees.

Each dialogue tree is represented by an instance of the *DialogueTree* class with attributes *id*, *name*, and *components*. *components* is a list containing instances of the *Component* class. Each *Component* instance has attributes *id*, *name*, and *neighbors*. *neighbors* is a list containing other *Component* instances and is used to store directed edges in the dialogue tree. A *Component* instance is always one of its subclasses: *Generation* or *Detection*. *Generation* instances represent generation components and have additional attributes *gen_class* and *examples*. The *gen_class* is the type of user inputs the component must respond to. *examples* is a list containing instances of class *GenerationExample*. Each *GenerationExample* instance has an *id*, *context*, and *response*. *Detection* instances represent detection components and have additional attribute *classes*. *classes* is a list containing instances of class *DetectionClass*. Each *DetectionClass* instance has an *id*, *det_class*, and *examples*. *det_class* is a specific class name, and *examples* is a list containing instances of class *DetectionExample*. A *DetectionExample* instance has an *id* and *example*.

Each component must exist within one dialogue tree, but components can be replicated within the same tree. The ids throughout the model are used to uniquely identify editable elements.

Implementation Details: Validations

Before processing each API request, a corresponding validation function for that API is called. These validation functions prioritize checks that prevent basic code errors and steer the user in the right direction. The primary validation categories are as follows:

1. Required request fields are present
2. Required request values have the correct data type
3. Specified items with ids exist
4. Specified edges between components exist
5. Conversation history between student and chatbot is formatted correctly

Validations more complex than this, such as ensuring the dialogue tree structure is logical, are up to the chatbot designer. To facilitate easier debugging, custom error messages and status codes are returned in API responses when requests fail a validation function.

Implementation Details: Detection Prompt Engineering

Crafting a quality detection prompt was key in making detection components accurate. Several design decisions here were taken from the OpenAI documentation on implementing LLM classifiers. First, instructions and few-shot examples are best for classification tasks. The quality and quantity of examples is also important, and a low temperature is best to ensure the model outputs deterministic, confident class predictions. Lastly, "text-davinci-003" was used because it is OpenAI's best LLM for text completion (which is what the classifier must do).

To explore different prompts, 36 example user inputs across 3 different sample scenarios were created. Then, three different prompt formats were tested on all 36 examples. Each prompt contained instructions regarding the classification task. However, the examples were either provided (few-shot) or not (zero-shot) and context from the conversation history was either provided (context) or not (no context). The results are summarized below.

Prompt Format	Accuracy	Hallucinations	Comments
Zero-shot with no context	21/36	0/36	Hallucinations not an issue; without context, hard to distinguish roles in conversation
Zero-shot with context	31/36	0/36	Context helped identify conversation roles, but pulling context from the frontend would be hard
Few-shot with no context	34/36	0/36	Examples were extremely helpful even though the current context is not present

Table 1: Summary of findings from detection prompt engineering experiments.

As the table shows, the prompts containing either few-shot examples or context performed the best. Including the live context in our system would have been difficult, because Social Media

TestDrive is a simulated social media platform with a variety of different features such as comments, posts, likes, replies, and more. Pulling this information from the frontend to help explain the context of the user's messages would be difficult. With generic few-shot examples, however, a similar performance boost was witnessed. Since adding few-shot examples was much easier in this prototype system, the "few-shot with no context" approach was used. The final prompt format is shown below.

```
Classify the user inputs into one of the following categories: [category_1], . . ., [category_n]
Input 1: [example input 1]
Category 1: [example category 1]
Input 2: [example input 2]
Category 2: [example category 2]
.
.
.
Input 2n: [example input 2n]
Category 2n: [example category 2n]
Input 2n+1: [input text to be classified]
Category 2n+1:
```

Figure 5: Outline of detection prompt format, where each class has roughly two few-shot examples. It begins with an instruction that lists the classes in the detection component. Next, the few-shot examples stored in the component are listed. Finally, the current user input to be classified is provided.

Aside: Generation Prompt Engineering

Developing the generation prompt required a similar set of experiments that were done by another member of the project. After experimentation, the best chatbot responses were achieved by including an instruction, the generation class, few-shot examples, conversation history, and the new input to respond to.

Implementation Details: APIs

All of the aforementioned functionality is contained within 30 Flask APIs. These APIs are the only way in which the backend can be interacted with and are all described in the table below. Note that failure responses are not included, because the status codes and error messages vary depending on the error that is found.

Method and URL	Request and Success Response Body	Functionality

POST /dialogue	Request body:
	{
	“name”: “dialogue tree name”
	}
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “tree id”
	}
	}
		Create dialogue tree

GET /dialogue/<dt_id>	Request body:
	None
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “tree id”
	“name”: “dialogue tree name”
	“components”: [
	“first id”,
	“second id”
	]
	“edges”: [
	{
	“start”: “first id”,
	“end”: “second id”
	}
	]
	}
	}
		Get dialogue tree

DELETE /dialogue/<dt_id>	Request body:
	None
	
	Success response body:
	{
	“success”: true
	}
		Delete dialogue tree

PUT /dialogue/<dt_id>/name	Request body:
	{
	“name”: “new dialogue tree name”
	}
	
	Success response body:
	{
	“success”: true
	}
		Edit dialogue tree name

POST /dialogue/<dt_id>/edge	Request body:
	{
	“start”: “start component id”
	“end”: “end component id”
	}
	
	Success response body:
	{
	“success”: true
	}
		Create dialogue tree edge

DELETE /dialogue/<dt_id>/edge	Request body:
	{
	“start”: “start component id”
	“end”: “end component id”
	}
	
	Success response body:
	{
	“success”: true
	}
		Delete dialogue tree edge

POST /dialogue/<dt_id>/generation	Request body:
	{
	“name”: “generation component name”
	}
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “component id”
	}
	}
		Create generation component

GET /dialogue/<dt_id>/generation/<gc_id>	Request body:
	None
	
	Success response body:
	{
	"success": true,
	"data": {
	"id": "component id",
	"name": "component name",
	"class": "class name",
	"examples": [
	{
	"id": "example id",
	"context": "example context",
	"response": "example response"
	}
	]
	}
	}
		Get generation component

DELETE /dialogue/<dt_id>/generation/<gc_id>	Request body:
	None
	
	Success response body:
	{
	"success": true
	}
		Delete generation component

PUT /dialogue/<dt_id>/generation/<gc_id>/name	Request body:
	{
	"name": "new component name"
	}
	
	Success response body:
	{
	"success": true
	}
		Edit generation component name

PUT /dialogue/<dt_id>/generation/<gc_id>/class	Request body:
	{
	"class": "new generation class name"
	}
		Edit generation component class

	Success response body:
	{
	“success”: true
	}	

POST /dialogue/<dt_id>/generation/<gc_id>/example	Request body:
	{
	“context”: “example context”,
	“response”: “example response”
	}
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “example id”
	}
	}
		Create generation example

DELETE /dialogue/<dt_id>/generation/<gc_id>/example/<ex_id>	Request body:
	None
	
	Success response body:
	{
	“success”: true
	}
		Delete generation example

PUT /dialogue/<dt_id>/generation/<gc_id>/example/<ex_id>	Request body:
	{
	“context”: “new example context”,
	“response”: “new example response”
	}
	
	Success response body:
	{
	“success”: true
	}
		Edit generation example

POST /dialogue/<dt_id>/generation/<gc_id>/copy	Request body: None	Copy generation component

	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “component id”
	}
	}	

POST /dialogue/<dt_id>/detection	Request body:
	{
	“name”: “detection component name”
	}
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “component id”
	}
	}
		Create detection component

GET /dialogue/<dt_id>/detection/<dc_id>	Request body:
	None
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “component id”,
	“name”: “component name”,
	“classes”: [
	{
	“id”: “class id”,
	“class”: “class name”
	}
	]
	}
	}
		Get detection component

DELETE /dialogue/<dt_id>/detection/<dc_id>	Request body: None	Delete detection component

	Success response body:
	{
	“success”: true
	}	

PUT /dialogue/<dt_id>/detection/
<dc_id>/name
	Request body:
	{
	“name”: “new component name”
	}
	
	Success response body:
	{
	“success”: true
	}
		Edit detection component name

POST /dialogue/<dt_id>/detection/
<dc_id>/class
	Request body:
	{
	“class”: “class name”,
	}
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “class id”
	}
	}
		Create detection class

GET /dialogue/<dt_id>/detection/
<dc_id>/class/<cls_id>
	Request body:
	None
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “class id”,
	“class”: “class name”,
	“examples”: [
	{
	“id”: “example id”,
	“example”: “class example”
	}
	]
	}
	}
		Get detection class

DELETE /dialogue/<dt_id>/detection/<dc_id>/class/<cls_id>	Request body:
	None
	
	Success response body:
	{
	“success”: true
	}
		Delete detection class

PUT /dialogue/<dt_id>/detection/<dc_id>/class/<cls_id>/name	Request body:
	{
	“class”: “new class name”
	}
	
	Success response body:
	{
	“success”: true
	}
		Edit detection class name

POST /dialogue/<dt_id>/detection/<dc_id>/class/<cls_id>/example	Request body:
	{
	“example”: “class example”,
	}
	
	Success response body:
	{
	“success”: true,
	“data”: {
	“id”: “example id”
	}
	}
		Create detection class example

DELETE /dialogue/<dt_id>/detection/<dc_id>/class/<cls_id>/example/<ex_id>	Request body:
	None
	
	Success response body:
	{
	“success”: true
	}
		Delete detection class example

PUT /dialogue/<dt_id>/detection/<dc_id>/class/<cls_id>/example/<ex_id>	Request body:
	{
	“example”: “new class example”,
	}
		Edit detection class example

	Success response body:
	{
	"success": true
	}	

POST /dialogue/<dt_id>/detection/
<dc_id>/copy
	Request body:
	None
	
	Success response body:
	{
	"success": true,
	"data": {
	"id": "component id"
	}
	}
		Copy detection component

POST /dialogue/<dt_id>/detection/
<dc_id>/prompt
	Request body:
	{
	"messages": [
	{
	"role": "chatbot",
	"message": "first chatbot message"
	},
	{
	"role": "student",
	"message": "first student message"
	}
	]
	}
	
	Success response body:
	{
	"success": true,
	"data": {
	"response": "classification result"
	}
	}
		Classifies the most recent student message in *messages* using the specified detection component. Calls GPT-3 using the Azure OpenAI Service.
		
		This API is just for testing purposes.

POST /dialogue/<dt_id>/generation/ <gc_id>/prompt	Request body: { "messages": [{ "role": "chatbot", "message": "first chatbot message" },	Generates a response to the most recent message in *messages* using the specified generation component. Calls GPT-3 using the Azure OpenAI Service.

	{
	"role": "student",
	"message": "first student message"
	}
	]
	}
	
	<u>Success response body:</u>
	{
	"success": true,
	"data": {
	"response": "chatbot message"
	}
	}
		This API is just for testing purposes.

POST /dialogue/<dt_id>/chat/<c_id>	<u>Request body:</u>
	{
	"messages": [
	{
	"role": "chatbot",
	"message": "first chatbot message"
	},
	{
	"role": "student",
	"message": "first student message"
	}
	]
	}
	
	<u>Success response body:</u>
	{
	"success": true,
	"data": {
	"responses": [
	"chatbot message"
	],
	"next_id": "next detection component id" or "exit"
	}
	}
		Traverses the dialogue tree starting at the specified component, which should be a detection component in this prototype. Returns the generation component responses and id of the next component in the tree (or "exit" if at a leaf).
		
		This API combines the functionalities of the previous two APIs.

Table 2: All API methods, URLs, request and response body formats, and functionalities.

Demo

The following screenshots demonstrate what a real dialogue tree might look like for our use case, which is Social Media TestDrive. They also show a sample conversation between a user and the chatbot. Note that the frontend connecting to the APIs was done by other group members.

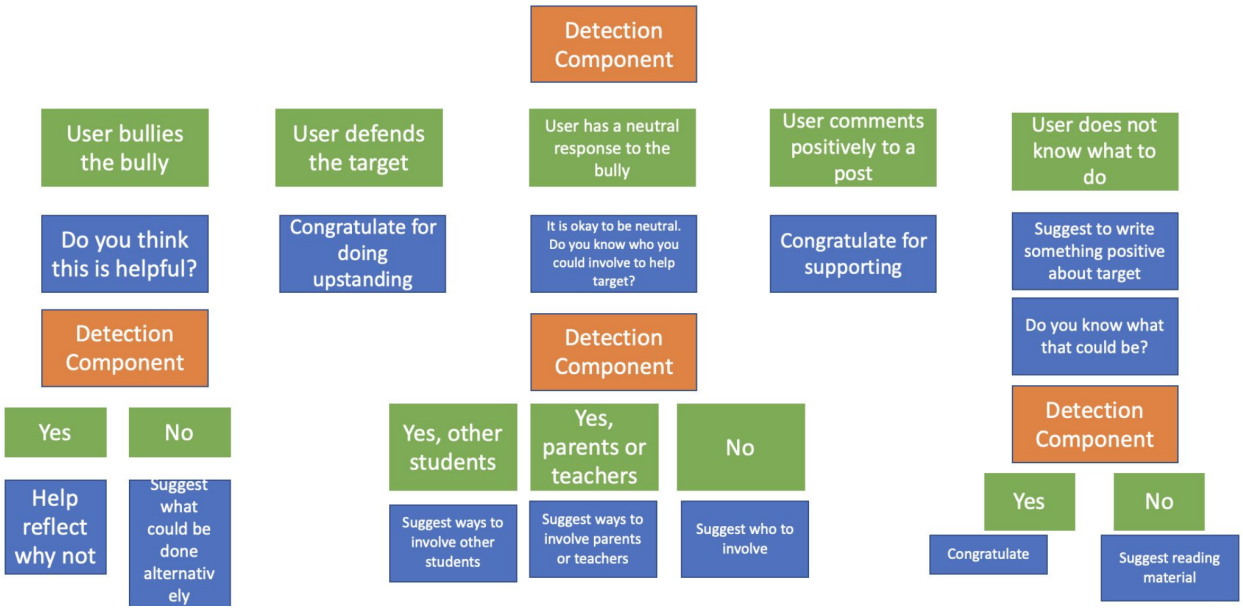

Figure 6: An example dialogue tree. The orange blocks are detection components, the green blocks are the detection component classes, and the blue blocks are generation components.

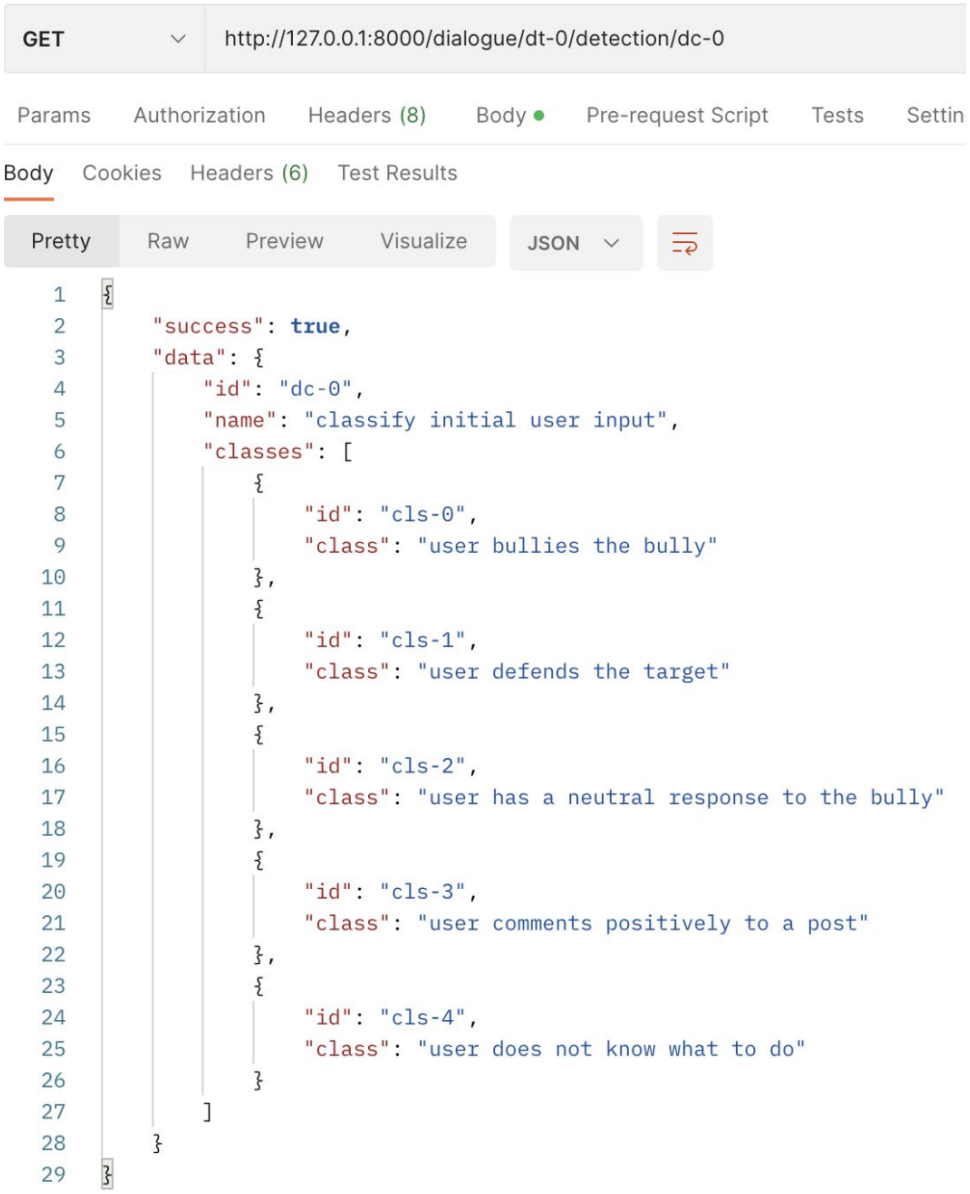

Figure 7: API call retrieving the first detection component at the top of the dialogue tree. This component contains the five classes from the figure.

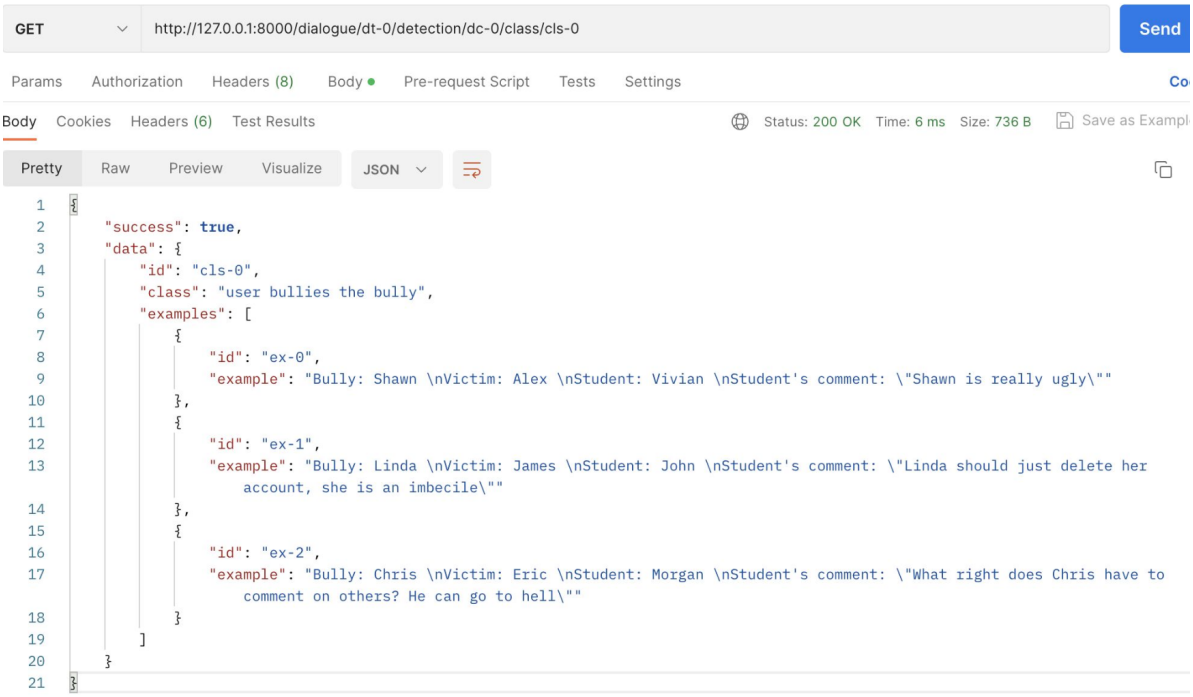

Figure 8: API call retrieving the first detection class in the first detection component. This component contains three few-shot examples of the user bullying the bully.

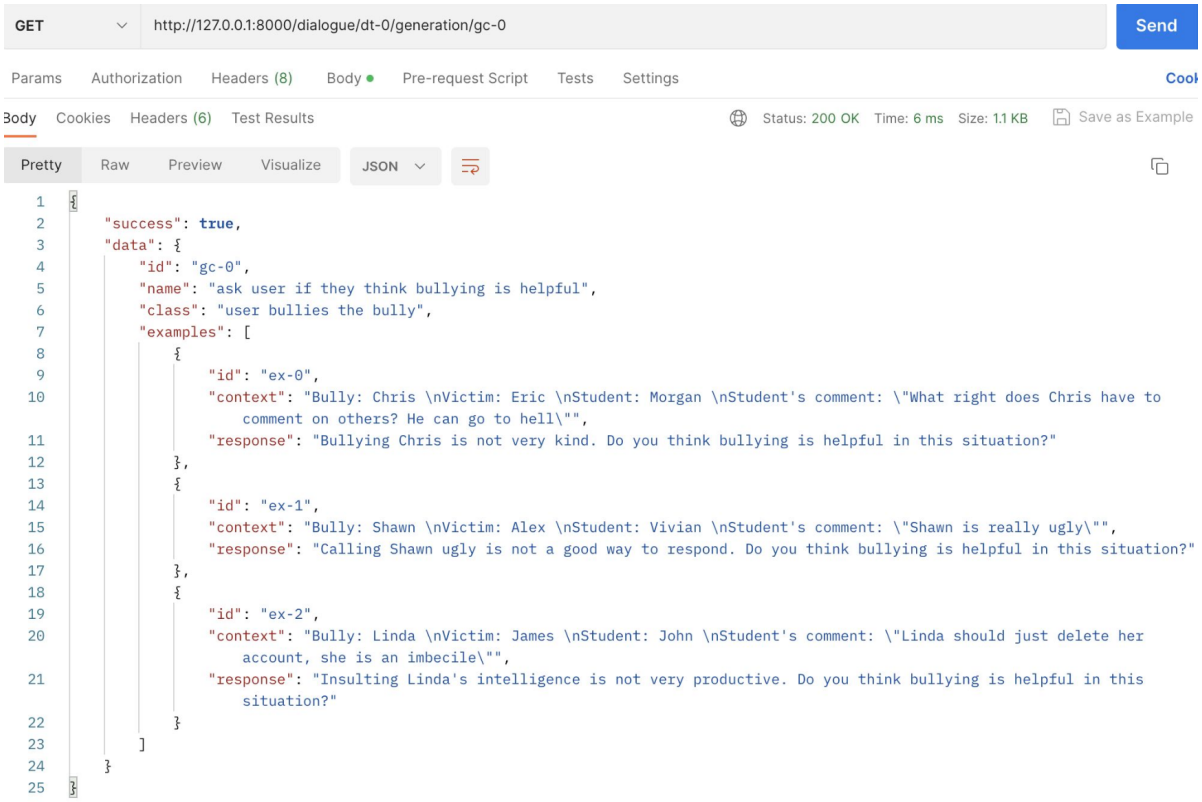

Figure 9: API call retrieving the generation component to respond to a user bullying the bully. This component contains three few-shot example responses.

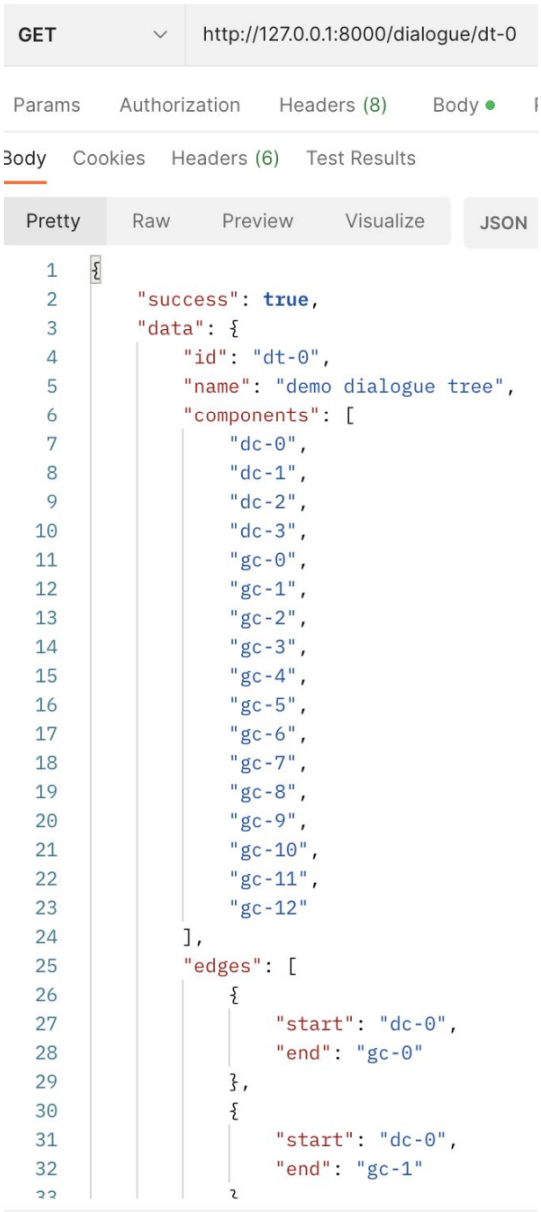

Figure 10: API call retrieving the fully constructed dialogue tree. This call retrieves the component ids and the edges between them.

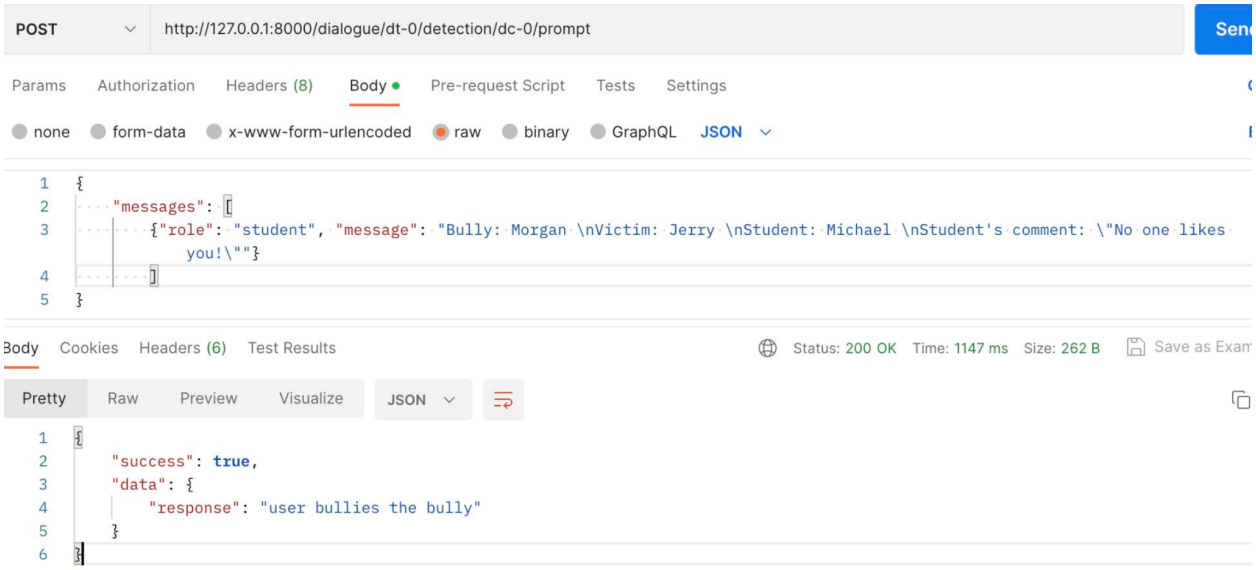

Figure 11: Detection API call to perform classification using the first detection component. The most recent user message in the input gets classified as "user bullies the bully".

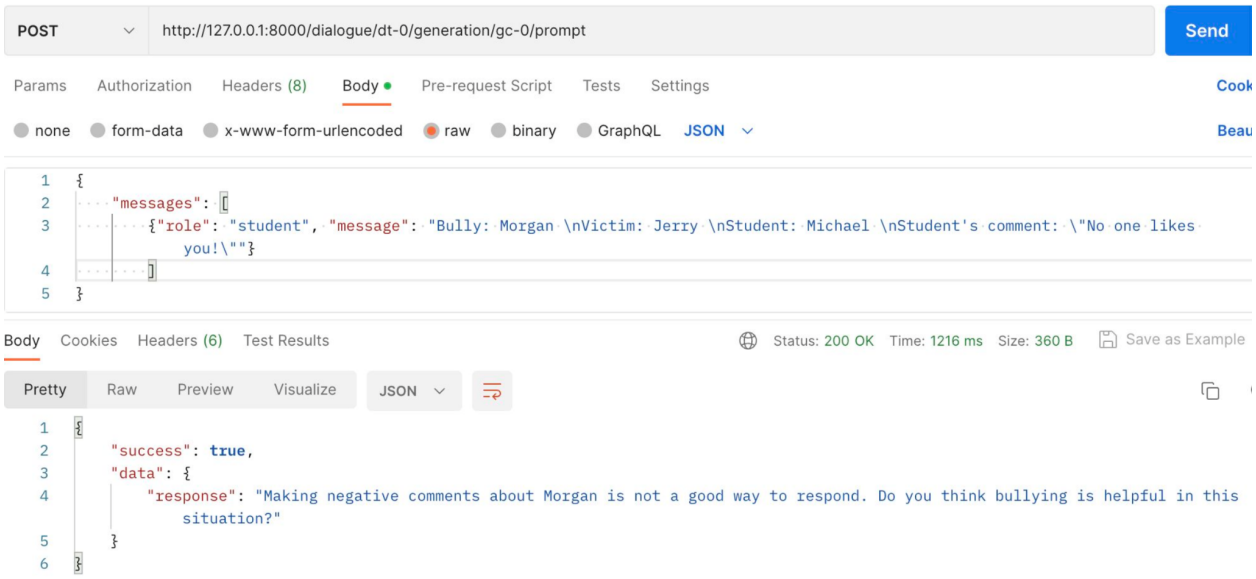

Figure 12: Generation API call to generate a chatbot response to the user bullying a bully. The chatbot responds to the most recent message, which in this case is from the student.

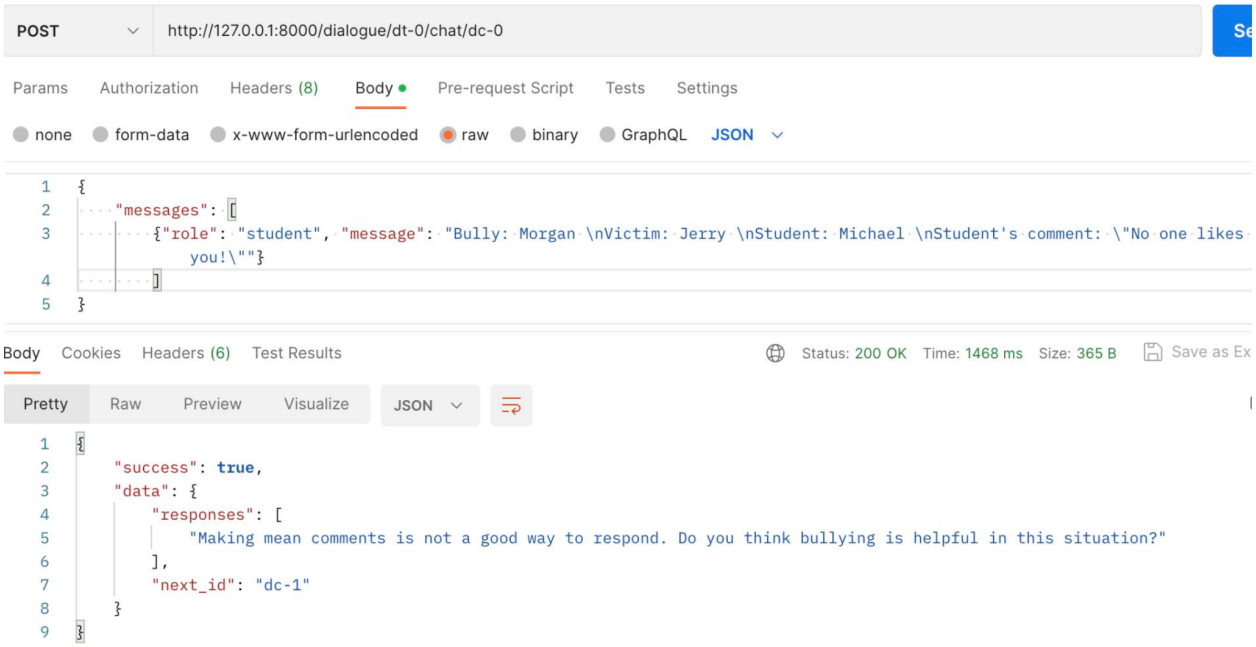

Figure 13: Chat API call to traverse the dialogue tree. The most recent user message gets classified as "user bullies the bully", and the chatbot responds appropriately using the correct generation component.

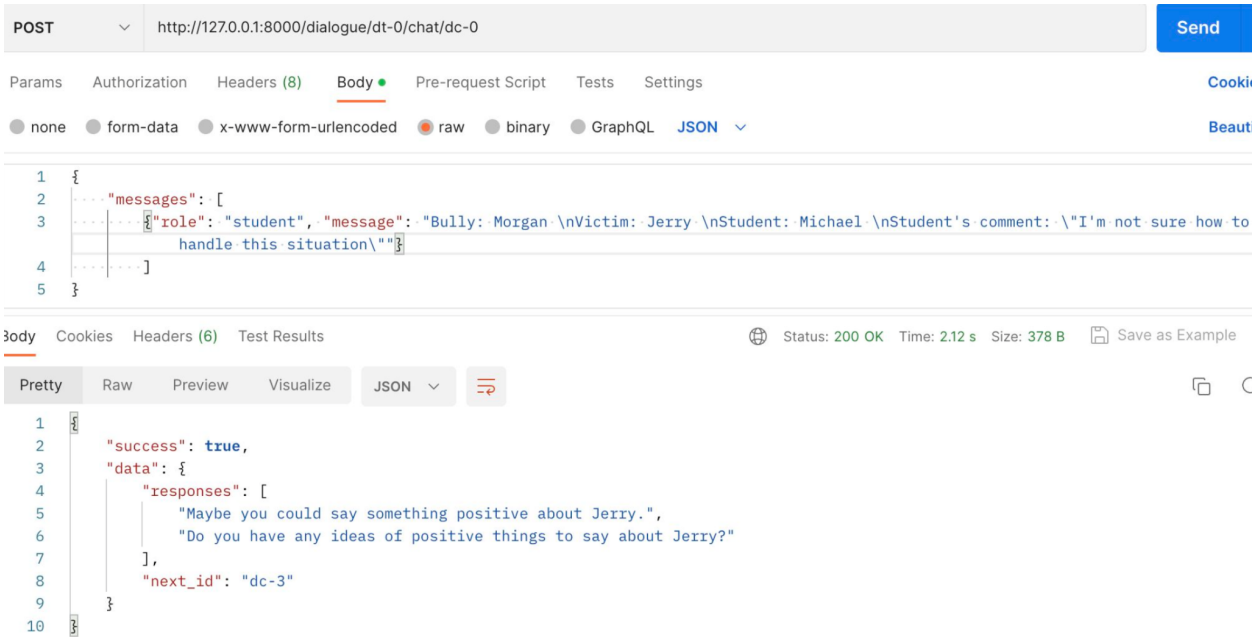

Figure 14: Turn one in an example conversation. The user asks for help, and the chatbot responds with two messages giving suggestions.

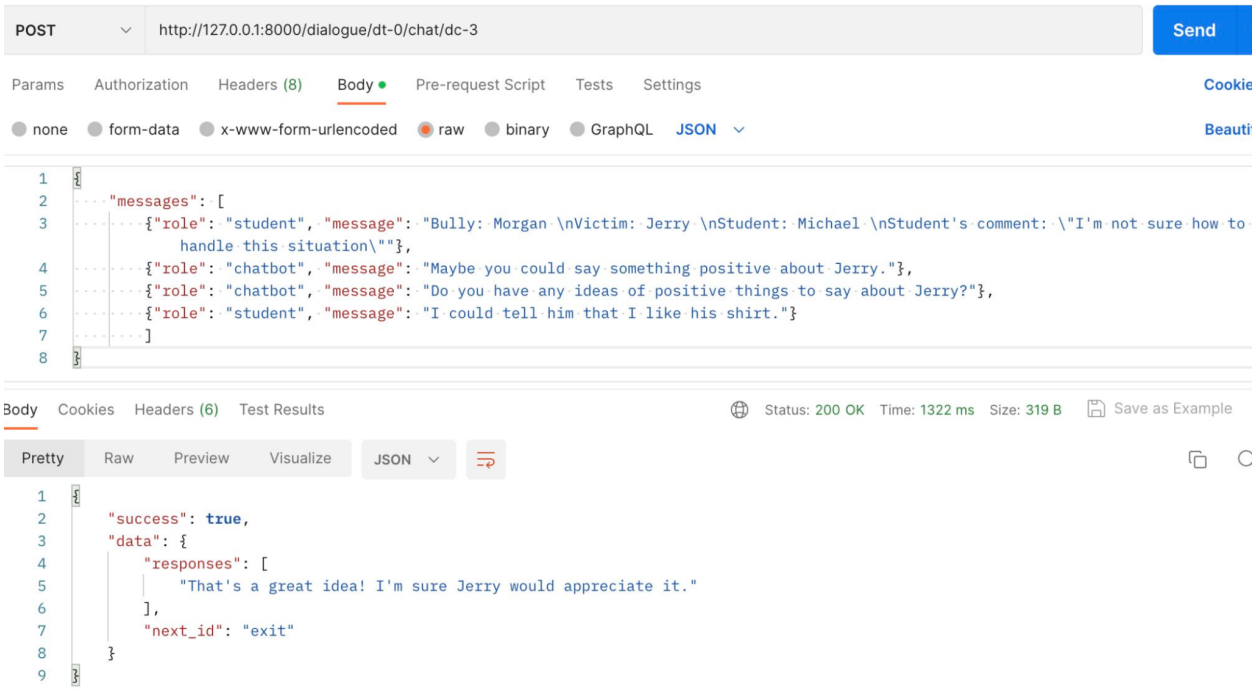

Figure 15: Turn two in an example conversation. The previous chatbot responses were added to the conversation history, and then the user sends their next message to the next detection component. The chatbot responds again, and now the end of the dialogue tree has been reached.