Dialogue Distillery: Crafting Interpolable, Interpretable, and Introspectable Dialogue from LLMs

Ryan A. Chi, Jeremy Kim, Scott Hickmann, Siyan Li, Gordon Chi, Thanawan Atchariyachanvanit, Katherine Yu, Nathan A. Chi, Gary Dai, Shashank Rammoorthy, Ji Hun Wang, Parth Sarthi, Virginia Adams, Brian Y. Xu, Brian Z. Xu, Kristie Park, Steven Cao, and **Christopher D. Manning**
Stanford NLP
https://stanfordnlp.github.io/chirpycardinal/

Abstract

We present the third iteration of **Chirpy Cardinal**, an open-domain dialogue agent developed for the Alexa Prize Socialbot Grand Challenge 5 (SGC5) competition.[1] In 2023, while pure large language model (LLM) chatbots have superior breadth, flexibility, and fluency compared to traditional dialogue trees, they have high latency and are not factual, controllable, or maintainable, limiting their effectiveness as dialogue systems. To address these weaknesses while preserving the strengths of a neural system, we propose a chatbot architecture that involves *distilling* a language model into a *symbolic structure*. Toward this end, we propose the following innovations: (a) **CAMEL**, a domain-specific language and graphical interface for chatbot development, (b) **Goldfinch**, which automatically generates nodes in a conversational graph using an LLM, and (c) **Kingfisher**, which reduces latency and enables use of larger neural models by predicting future user utterances and precomputing responses to them. In deployment, we find that these methods substantially improve conversation quality. We release our open-source code and hope it serves as a useful toolbox for custom chatbot development.

1 Introduction

Recently, neural large language models (LLMs) have produced great progress in the conversational domain and have been brought to the fore of public attention. With the rise of ChatGPT, products across a wide range of industries have been reconfigured to offer LLM-backed chat interfaces (OpenAI, 2023; Thoppilan et al., 2022). Compared to traditional chatbot setups based on dialogue trees (Galitsky and Ilvovsky, 2017; Medeiros et al., 2019), LLMs have the advantages of being broad, flexible, and fluent: they enable conversations across a breadth of domains, handle novel situations gracefully (Huang et al., 2022), and effortlessly generate idiomatic text. However, as has been observed in many currently deployed systems, these models also have glaring weaknesses that limit their effectiveness as the sole backend of a chatbot. In particular:

1. LLMs are not factual: They often hallucinate, contradict themselves when asked rephrased versions of the same question, give biased answers, and generate plausible-sounding but incorrect statements (Elazar et al., 2021; Shen et al., 2023; Min et al., 2023).
2. LLMs are not controllable: A good conversational agent needs to be more than just fluent; it must also be responsive to steering away from undesirable topics. Unfortunately, it is notoriously difficult to control LLMs in this way (Zhou et al., 2022).
3. LLMs are risky: Language models can degenerate into producing toxic output, even if the prompt text is not toxic (Dhamala et al., 2021; Gehman et al., 2020).

[1] https://www.amazon.science/alexa-prize/socialbot-grand-challenge

4. LLMs have high latency: LLMs require enormous amounts of memory and compute, and they continue to grow in size each year. Latency is especially important in the conversational domain; it is difficult to decode with sufficiently low latency even when using methods like quantization, distillation, or specialized hardware (Hsieh et al., 2023; Gudibande et al., 2023).
5. LLMs are not maintainable: Language models are generally neither modular nor editable, making it difficult to patch bugs or address problems that are exposed post-deployment (Yao et al., 2023).

Earlier dialogue systems often relied on well-structured, symbolic dialogues (Galitsky and Ilvovsky, 2017; Allen et al., 1996). Why is symbolic dialogue attractive? For a relatively small organization, symbolic dialogue is (1) more introspectable, (2) definitively interpretable, and (3) guaranteed to be non-toxic, as well as being easy to collaborate on and update. This addresses the traditional problem of deep learning being uninterpretable (Castelvecchi, 2016). But the amount of human labor it takes to write a dialogue tree that responds appropriately to all possible human utterances is excessive.

Yet, perhaps a happy medium is possible. What if we could extract some of the flexibility of a neural language model into a smaller, more symbolic form? In particular, we would like its ability to respond fluently to a wide array of topics, while still being controllable, editable, and stateful. We believe that, given a model of this form, we could interpolate between neural expressivity and prewritten commonsense, gaining a lot of flexibility while still having the ability to control the direction of the conversation.

This is precisely what we propose and build in **Chirpy Cardinal 3.0**, a novel chatbot architecture that seeks to address these aforementioned weaknesses while preserving the breadth and flexibility of an LLM-backed system. Our key idea is to distill the language model into a symbolic structure, capturing the controllability, maintainability, and low latency of a symbolic system without requiring large amounts of hand-coding in a traditional setup. Toward this goal, we develop:

1. A chatbot development framework, which includes **CAMEL** 🐪, a domain-specific language and state-tracking system, and **Bluejay** 🐦, a graphical interface for chatbot development.
2. **Goldfinch** 🐤, a set of LLM-backed tools to automatically generate nodes in a conversational graph and navigate between them in conversations. The resulting distilled system is highly interpretable and manually editable while preserving the breadth of a neural system.
3. When the automatically generated nodes are insufficient, we use a neural LLM as a backup. To reduce latency and enable larger models, we develop **Kingfisher** 🐦, a "pre-firing" method that predicts future user utterances and precomputes responses to those utterances. This method allows us to take advantage of the time the user spends thinking to run the LLM in the background, reducing latency and producing substantial gains in dialogue quality.

We showcase an example dialogue illustrative of our system in Figure 1. In the remainder of this paper, we describe our system in detail, as well as the lessons learned when deploying it as part of the Alexa Prize SGC5 competition (Johnston et al., 2023). Upon the conclusion of SGC5, we will open-source our code, and we hope that it will serve as a useful toolbox for custom chatbot development.[2]

2 Conversational Modeling

Over the course of the SGC5, we built a library for social dialogue from scratch. Compared to previous libraries such as CoBot[3] and the previous Chirpy codebase (Chi et al., 2022), our library entails several advantages in terms of scalability, interpretability, maintenance, latency, and dialogue flexibility:

1. The CAMEL language is declarative (see Section 5), vastly improving **interpretability**, **maintenance**, and **flexibility**.
2. CAMEL decomposes broad conversation topics like music into finer topics, which induces rich conversation graphs and increased **dialogue flexibility** (Section 2).

[2] https://stanfordnlp.github.io/chirpycardinal/

[3] https://www.amazon.science/blog/ai-tools-let-alexa-prize-participants-focus-on-science

LAUNCH begins the conversation if we have not spoken with the user before (otherwise, we greet them by name).

FOOD__INTRO prompts first-time users about food, a subject of universal interest.

Entity Linker identifies *pizza* as the new entity.

The *entry condition* (Section 5) to enter <SPECIFIC__Pizza> is that `Current Topic` equals `"Pizza."`

<SPECIFIC__Pizza> defines one turn of neural follow-up (Section 2.2) to its question, in order to reduce the speed of topic transition.

Neural generation provides a flexible response.

Factoids from GPT-3 are prompted with a knowledge base, and checked manually to ensure factuality.

Pre-computed transitions (Section 2.5) create a sense of conversational continuity.

SPECIFIC__* nodes, created via *Goldfinch* (Section 3), allow us to preanticipate common responses (*e.g.* art museums in NYC).

The **Personal Issues** supernode (Section 2.2) provides a comforting statement.

Figure 1: An example dialogue. To respect users' privacy, this is a conversation created by a team member (not with a real user). We model dialogue as a series of supernodes (left side), whose topics are parsed and tracked by a neural entity linker. Most dialogue is *distilled* from a large language model (GPT-3; Brown et al. (2020)) into a symbolic form, while regex-based NLU is used to find conversational paths. To follow up on such conversational statements, we use a smaller ***neural generator*** (BlenderBot-3B; Roller et al. (2021)) to dynamically generate neural dialogue.

Turn 0	I like to jog. Jogging is such a fun activity! / By the way, what's your favorite food?
Turn 1	I love potatoes. That sounds delicious. / Have you ever tried them with hot sauce?
Turn 2	No, I've never tried that before. You should try it! It's delicious.
Turn 3	Really? I cannot handle spice though. You can improve your spice tolerance! / Should I tell you how?

Figure 2: Dialogue turns from a sample conversation (created by a team member). Purple represents the follow-up phase and blue, the prompt phase. Note that Turn 2 has no prompt phase.

3. We distill LLMs through Goldfinch (Section 3), allowing the bot to **scale** to an arbitrary number of topics.
4. We utilize pauses in between turns to run even larger language models, without sacrificing **latency**, through Kingfisher (Section 4). Additionally, framework-related latency is low (<100ms).

2.1 Modeling Conversations: the prompt and follow-up phase

Consider the sample conversation in Figure 2. We model bot responses as a combination of at least one of the following phases:

1. A *follow-up* phase, where the bot responds to what the user just said. The follow-up phase is easiest to model when the user mentions an entity, simplifying natural language understanding (NLU). Abstract, non-factual subjects are the most difficult to prepare responses for, but neural models often respond well to these non-factual topics.
2. A *prompt* phase, where the bot takes some initiative and/or initiates a new topic. Although we typically require a follow-up phase (to sufficiently acknowledge the user's response), the prompt phase may not be needed if either the follow-up phase or user provides initiative, as per the principles of mixed-initiative dialogue (Horvitz, 1999).

2.2 Supernodes

To model each $<$ prompt, follow-up $>$ pair, we create a large number of *supernodes*, akin to the tree nodes of a classical dialogue tree (Weizenbaum, 1966), except these are now part of a *conversational graph*. Each supernode thus handles dialogue over two turns: the prompt phase of the first turn, and the follow-up phase of the next. Importantly, each supernode is **modeled separately from the rest** and is designed to not make assumptions about the surrounding conversational situation beyond its entry conditions, allowing for the composing of supernodes in arbitrary order. Given a conversational state, a supernode **returns a bot utterance and updates to the bot's state**.

We treat each supernode as handling a small section of conversation: one supernode contains at least one prompt and a set of responses, which we term *subnodes*. In turn, each subnode represents a possible response to the user statement, allowing the socialbot to provide specific responses.

More precisely, a supernode's prompt phase contains:

- ***NLU***, which identifies key features of the user utterance, such as sentiment and keywords. In CAMEL, NLU is referred to as "***flags***" and applies to the current turn only, i.e., we reset flags after each turn. The appendix contains more details about NLU.
- A set of ***entry conditions*** required for the supernode to activate, or prompt. For example, the **<FOOD_intro>** node, which asks the user about their favorite food, only makes sense if the user's favorite food is not yet known. Entry conditions are defined using CAMEL's predicate syntax.
- One or more ***prompt(s)*** to be used during the prompt phase. Each prompt can define its own entry conditions, allowing for specialized prompts.

The follow-up phase contains:

- NLU and entry conditions, as in the prompt-phase.
- A set of ***takeover entry conditions*** required for the node to take over in the follow-up phase. We use this to deal with unexpected situations not anticipated during the prompt phase, such as high-initiative user actions, return questions back to the user, or the user requesting navigation.
- ***Subnode(s)*** to generate the follow-up phase's response. Each subnode can define its own entry conditions. A supernode typically defines multiple subnodes, with each pertaining to a certain response that a user might provide.
- (Optionally) A request for some number of post-hoc neural turns (see below).

Moreover, each section (prompt, subnode, and the prompt phase/followup phase altogether) can update **state**, which keeps track of the state of the conversation. Our state allows for the integration of things we know about the user, such as their favorite food or pets later in the conversation.

Supernodes are defined in CAMEL (Section 5).

Post-hoc Neural Followups An excessive amount of prompt/follow-up structured dialogue often results in a feeling of "whiplash," as the bot fails to give the user sufficient initiative (Horvitz, 1999), leading the conversation to devolve into what one end user termed a "question and answer session [that is] clearly run by the bots."[4] We balance structured dialogue with *post-hoc neural followups*, individually declared by each node, where our neural generator responds directly for a certain number of turns. This balance lets us utilize neural dialogue's fluency, while maintaining control over the conversation, and it effectively equates to a *speed-specificity tradeoff*: using more neural reduces the extent to which the socialbot takes initiative over the conversation, which results in a more comfortable but less precise experience.

2.3 Conversational Flow

At any given point, many possible supernodes are available. Thus, many possible orders of conversation are possible; instead of a conversational tree, we have a **conversational graph**, creating a combinatorial explosion of possible dialogue paths both within a topic (see Figure 3) and between topics (see Figure 14).

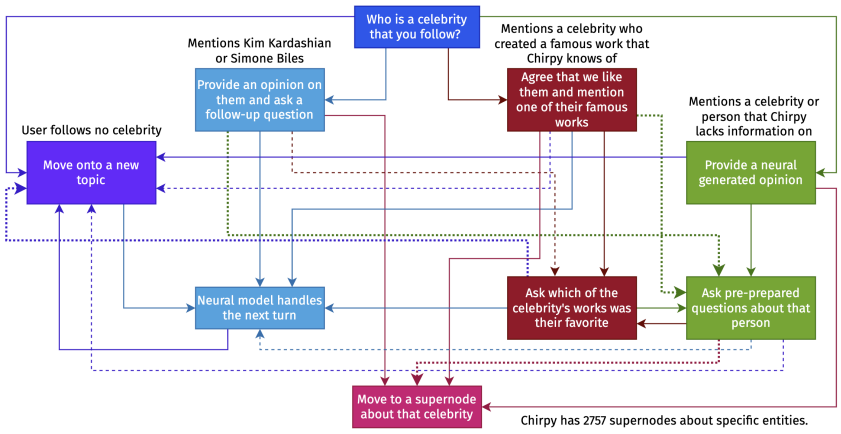


Figure 3: The conversation graph during an example discussion about celebrities with Chirpy. Due to the flexible condition-based system, almost any node can follow a given current node; we give nodes that result in more consistent dialogue flows higher weight through our ranking scheme.

To decide which supernode to pick next, we track the *current topic*, the current subject of the conversation. Our flexible dialogue system allows storing attributes such as the user's knowledge and thoughts about the current topic on the topic object itself. Following the Gricean maxim of quantity (Grice, 1975), we weight possible nodes by their level of *specificity* to the current situation, as seen

[4] https://www.amazon.com/Amazon-Alexa-Prize-Socialbots/dp/B06XYTWTX9

in Figure 4. Specifics regarding this weighting are in the appendix. This scheme naturally leads to a convincing and flexible conversation: we start with extremely specific conversational nodes and as our initiative recedes, we move to increasingly general nodes and allow the user to express their opinions about the topic (see Figure 4).

2.4 Entities

In Chirpy, all objects – whether actors, foods, instruments, or otherwise – are represented through a single framework: entities. By design, entities store information about the user in a structured fashion: as it learns more about the user, Chirpy saves key pieces of information by updating user entities like `favorite food` or `favorite place`. We automatically entity-link entities using a BERT-based entity linker (Broscheit, 2019; Chi et al., 2022).

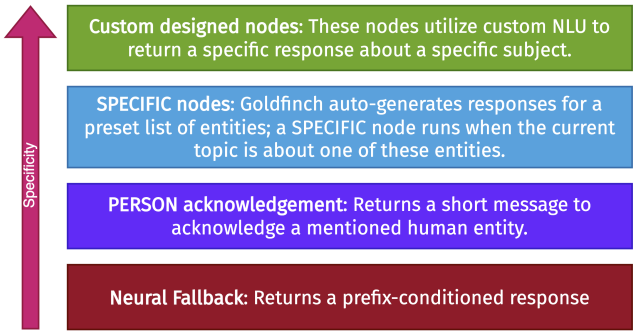


Figure 4: Common classes of nodes, with higher-up nodes being more specific.

Much like objects in OOPs, entities have properties (name, age, type) and inherit from the entity classes they fall into – for instance, `Paintings` fall under the `Artwork` class. Chirpy leverages the systematic structure of entities by prefilling properties of entities the user may want to discuss (with information extracted from Wikipedia). Using entity linking, Chirpy is able to automatically detect entities from a user utterance, cast them to the appropriate type, and load all the initial properties – all well before the user *explicitly* asks to talk about that specific entity.

Through this system, supernode entry conditions simplify to whether a relevant entity has been mentioned (either in the current user utterance or previously in conversation). Which entities are relevant is a matter of inheritance: if the current topic is a celebrity like Taylor Swift, it is relevant to any supernode which requires a celebrity or person entity to activate. Following this example, supernodes that require the current topic to be a celebrity have a higher chance of activating, since `Taylor Swift` inherits more closely from the `Celebrity` entity class than the broader `Person` class.

2.5 Smooth Transitions

The key to natural conversations is smooth transitions. Even though a chatbot system may be perfectly capable of a given topic in depth, it is impossible for its interactions to sound organic without clean and logical segues from one topic to the next. Without them, interactions come across as disfluent, abrupt, or even confusing (Dessalles, 2017).

Baseline: Transition Helper Statements By default, we randomly sample from a list of premade *transition helper statements* – generic connectors which are generalized enough to connect any conversation entity to any other. As an example, statements like “hm, so on another topic" are simple enough to tie together any two topics, but do they actually feel smooth?

Natural Transitions To address this issue, we create contextual transitions that are tailored to both the initial and final entities – which we call *bridging utterances*. For maximally natural conversations, we generate an entity to prioritize switching to as well as the transition to that entity. We leverage ChatGPT as a proxy for human-like conversations; we engineered a prompt to (1) predict the most likely supernode to switch to and (2) pre-generate short bridging utterances from the initial entity to the predicted entity. We use chain-of-thought prompting (Wei et al., 2022), asking GPT-3 (Brown et al., 2020) to first pick a new topic to move to from a restrained list, then generate an explanation for the transition, and finally produce a transition. We use ChatGPT-turbo 3.5 and a temperature of 1.1.

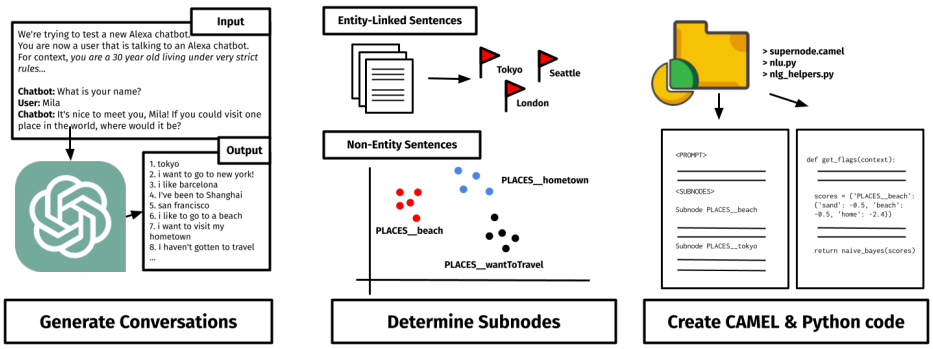


Figure 5: The Goldfinch pipeline. (1) *Generate Conversations*: We propose the use of large language models to generate potential user utterances and desired chatbot responses, using one-shot prompting alongside a range of personality descriptions in order to ensure diversity in output; (2) *Determine Subnodes*: We cluster subnodes with the help of a neural entity linker for entity-based utterances, while we cluster the remaining sentences with an affinity propagation model on the SentenceBERT embeddings; (3) *Create CAMEL Code*: Given user utterances and possible scenarios, we use GPT-3 to generate a sensible response to each subnode cluster, then generate a Naïve Bayes function to cluster non-entity user utterances at inference time.

3 Goldfinch : Distilling Large Language Models into Dialogue Graphs

Though rule-based chatbot systems have the advantage of lower latency compared to their transformer-based counterparts at inference time, such systems require many hours of human intervention and response generation in order to cover all conversational possibilities (Webb, 2000). Thus, we develop **Goldfinch**, a novel method to distill large chat-tuned LMs into structured dialogue (an overview of the Goldfinch process is depicted in Figure 5). This allows us to not only benefit from the contextual knowledge of large language models in specialized topic areas (Petroni et al., 2019) but also generate hundreds of sample conversations that can be easily modified or edited. The deterministic nature of the process allows human developers to easily take down or edit subnodes and their LM-suggested Chirpy responses. Note that distillation here is different from its traditional sense in ML literature.

3.1 Method

Consider the **Chirpy Cardinal** architecture as a directed graph: each supernode denotes a node in the graph (with multiple potential subnodes), and each directed edge a transition. Our **Goldfinch** process builds upon this architecture through two alternating rounds of generation. First, we perform **widening**, which generates new subnodes for a desired supernode; next, we perform **deepening** which creates new "follow-up" supernodes for each Goldfinch-generated subnode.

3.1.1 Widening

For the widening process, our pipeline must (1) generate sufficiently varied potential user responses, and (2) classify these utterances into general scenarios and suggest engaging responses accordingly. This can be broken down into a four-step process.

User Response Generation We sample a large number of responses from a large language model (LLM) given a conversation history up to the node we wish to update. To ensure diversity, we prompt with a variety of personas (persona prompting; Deshpande et al. (2023)) including age, education, and compliance to simulate users. Currently, we sample approximately 200 responses.

Clustering of Possible User Responses Many of the ∼200 generated responses are likely to fall into the same category; we want these to trigger the same response. We cluster these using the following approaches:

- If the user's response names an *entity*[5], we cluster all such responses together.
- If the user's response does not explicitly name a Wikipedia entity, namely more informal pop culture references (e.g., YouTubers, video games), we attempt to label a core *topic* using an LLM prompted to come up with an "entity that describes the response". If a *pseudo-entity* appears sufficiently many times, we cluster all of the corresponding responses together.
- We cluster the remaining entities using an affinity propagation model (Dueck, 2009) over their SentenceBERT (Reimers and Gurevych, 2019) embeddings. We use affinity propagation because it allows for a dynamic number of clusters to be created.

3.1.2 Bot Response Generation

For each clustered scenario, we generate a set of ideal Chirpy responses via prompting GPT-3. We choose the most frequently generated bot response to each clustered scenario as the ideal response. We improve safety and coherence in our prompt engineering by detailing virtual agent guidelines and appropriately defining whether a response should be generalized.

3.1.3 CAMEL Code Generation

Since we cluster subnodes in three ways, we have a different entry condition type for each of the subnodes determined via the previous clustering:

- **By entity linker:** we set the entry condition for each subnode to be a check on the `CurrentTopic` instance.
- **By neural entity linker:** we write an NLU function that checks for keywords in the user utterance, then triggers flags we then use as entry conditions for distinct scenarios.
- **By affinity propagation model:** we fit a Naïve Bayes model over one-hot word counts; we choose this approach due to its relative interpretability and ease of modification as a dialogue writer. This allows us to classify a sentence into a neural cluster at inference time.

At the end of the process, one typically generates about 50 new subnodes, which are automatically added to the corresponding CAMEL file.

3.2 Deepening

Though the widening process is sufficient in developing a single supernode, a conversation is not complete in just one turn. Thus, the next step is to lay the foundation for a follow-up discussion for each newly-generated subnode. We prompt a large language model to simulate follow-up questions for each subnode by including the entire conversational history in the context, specifying the level of answer specificity as appropriate. For specific entity subnodes, we request a *specific question* that shows interest and knowledge of the subject at hand; for non-specific subnodes, we ask for a generalized question that asks the user for further discussion on their opinion or the subject. Due to the exponential nature of deepening, our conversations are set by default to two turns of deterministic supernode questions, followed by a neural response; exceptions to this setting are granted based on experimental traffic.

3.3 Implementation

Language Model We use GPT-3 *text-davinci-003* (Brown et al., 2020) for all generations, other than deepening, for which we use ChatGPT-3.5 (chosen for its stronger performance in high-initiative discussions). We do not use any user data in our current pipeline. All our sample user utterances are generated, and no sample data is being sent to external services as a few-shot example. For LLM-based analysis of user conversations, we use *Falcon-40B* (Almazrouei et al., 2023).

Caching Running each step of the widening and deepening process is not only slow but computationally expensive. In order to avoid such complications, we introduce caching for both conversational histories and sample user utterances.

[5]Our system automatically identifies and tags all entities from Wikipedia.

Selective Deepening A major shortcoming of **Goldfinch** is the aforementioned exponential nature of deepening and the supernode-subnode structure: if every supernode has a lower bound of n designated subnode responses, then generating a unique follow-up supernode for a starting supernode will result in n^2 unique subnode responses after two turns of conversation, and n^k over k turns of conversation. While this process may allow Chirpy Cardinal to build up specific domain knowledge on a multitude of topics, the number of nodes becomes completely unmanageable at a certain depth of discussion.

As a result, our team utilizes selective deepening, which allows us to deactivate follow-up supernodes that are rarely triggered in our experimental traffic. At the same time, our analytics can also inform us on follow-up supernodes that we should widen for more fulfilling conversations. While this process is currently a manual process, we aim to further automate this in the future.

3.4 Results

So far, we have generated over 2,000 supernodes with Goldfinch. Most of these nodes are designed for deeper conversations on the most common entities identified by our entity linker in experimental traffic; these supernodes are defined in the form **<SPECIFIC__(entity)>** as introduced later in the paper. That being said, we also performed deepening on more general topics (i.e., discussions on hobbies, favorite places, and ways to relax); these supernodes were manually chosen by identifying shortcomings of our latest iteration of Chirpy. Overall, our Goldfinched nodes were received well by users: we observe that conversations that reach our active, more general Goldfinched supernodes (i.e., all supernodes reached from starting nodes **<PLACES__intro>** and **<RELAXATION__intro>**) scored an average rating of 3.59, compared to an average of 3.24 without, from March to June.

To avoid ratings-based noise, we also evaluated Goldfinch with GPT-3.5 (Chiang and yi Lee, 2023). We used these to both evaluate the impact of Goldfinch altogether and the impact of deepening conversations (i.e., recursively applying Goldfinch to produce deep conversations). For the former, we prompted the LLM to compare sample conversations from a Goldfinched-PLACES supernode and its previous handwritten version. The LLM preferred Goldfinched conversations with a p-value of 0.00002. For the latter, we compared conversations in our RELAXATION specialization that were deepened with Goldfinch to those by a neural model. The LLM preferred Goldfinched conversations with a p-value of 0.0098. One limitation of LLM-based comparisons is their limited consistency; to compensate for this, we evaluated each conversation three times using our methodology, and it gave the same preference every trial for nearly half of the conversations. Details are in Appendix J.5.

4 Kingfisher (Prefiring)

4.1 Motivation

Even with our novel distillation technique, unpredictable scenarios still necessitate the use of neural LLMs, where addressing latency concerns present a significant challenge, despite significant advances (Brown et al., 2020). To generate a mere 15-token response, a 40-billion-parameter model sharded over 4 NVIDIA T4 GPU's can take up to 4 seconds – far greater than the sub-second latencies users expect (Paranjape et al., 2020).

In response, we propose **Kingfisher** (Figure 6), a method for integrating significantly larger language models—with significantly higher latency—into a socialbot with zero cost. The key insight of our approach lies in leveraging the slowest component of the human-bot interaction pipeline: **the human user**. During a conversation, human users typically take 2-5 seconds to finish their response. Additionally, the Alexa agent itself takes even longer to read its response: up to 10 seconds if the utterance is long (Figure 7).

Kingfisher uses this downtime to predict human responses *ex-ante* and generate significantly higher-quality, responses for each follow-up, from a massively larger LLM. We call this approach "**prefiring**," as it allows us to compute reasonable completions before even having seen what the user has said. By comparing these pre-generated outputs to the actual user response using sentence similarity measurements, we can quickly identify the most appropriate bot follow-up.

In concept, Kingfisher draws parallels to branch predictors in CPU design – which predict the direction a branch will go before being evaluated. Our method is broadly system-agnostic in that it

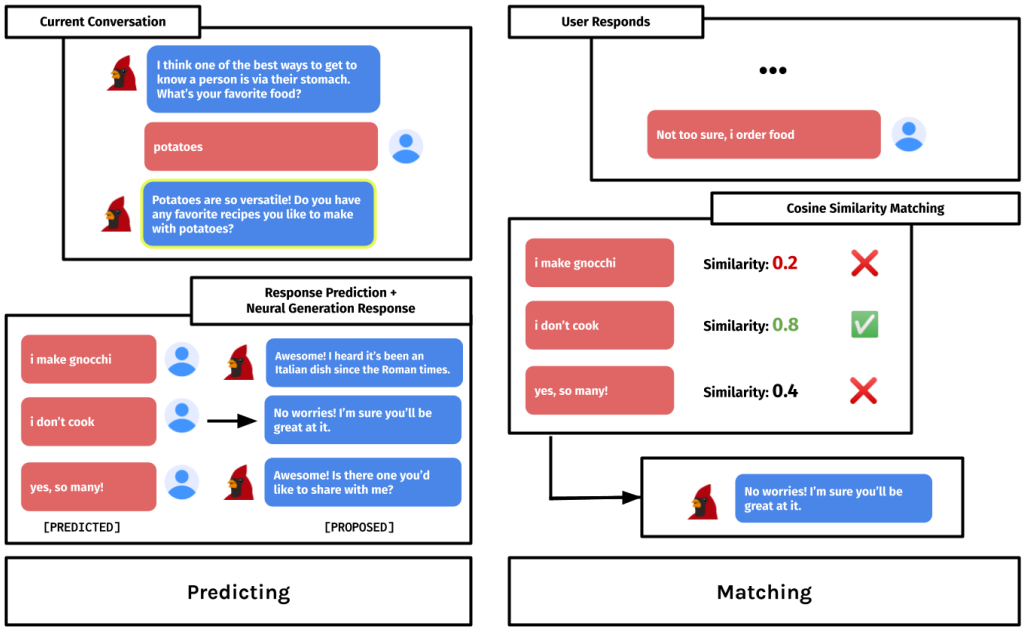


Figure 6: Prefiring consists of two phases. In the **Prediction** phase, we use the conversational history, including the last bot utterance, to anticipate what responses the user might produce. We then condition on each of these responses to generate responses from a much larger language model. In the **Matching** phase, which occurs on the next turn, we receive the actual real next utterance from the user and compute the similarity to our predicted completions. If one of the completions has sufficient similarity, we return it and thus produce a very strong neural response while incurring no time cost; otherwise, we defer to our default neural model.

can be applied for any chat-based application where one can (with some accuracy) predict the user's next response.

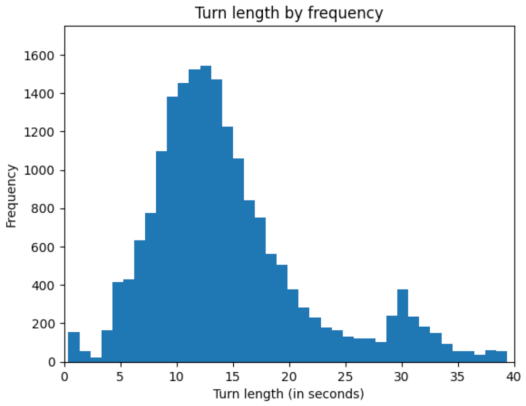


Figure 7: Histogram over seconds between end of the previous turn and beginning of the next turn.

4.2 Methods

As a general method, prefiring can be applied to any dialogue model—including entirely deep language models—where latency is a significant concern. In general, one needs a pair of two models: the *base model*, which has acceptable per-turn latency but worse response quality; and the *strong model*, which has better response quality but latency that exceeds the limits of a normal conversational turn.

Given a socialbot turn, the method proceeds as follows:

1. The previous turn of the socialbot finishes, and a response is sent to the user. Simultaneously, the response is sent to a **prefiring server**, which operates asynchronously (i.e. after the response is sent).
2. The prefiring server computes a set of **candidates** that the user might respond with. Ideally, this should cover as much of the probability space of user responses as possible.
3. The conversation history, extended with each candidate, is used to **generate candidate responses** from the strong model.
4. Once the actual observed user response for the next turn is received, we **compare** it under a specified similarity metric against the pre-computed candidates. If any candidate has high enough similarity, we return the corresponding candidate response, incurring only the latency of the similarity search. If not, we defer to the base model as usual, incurring only the base model's latency.

Notably, all generation occurs between the end of the previous bot reponse and receiving the next turn's user response. Since this time (on the order of 5 to 15 seconds) enormously dwarfs generation time (at most 3-4 seconds), one actually only pays a *negative* latency cost, as we incur only the lookup time.

4.3 Technical Notes

For candidate generation, we apply a BlenderBot-3B model, trained to imitate low-initiative users. We trained the agent on two weeks of data, selecting for low-initiative users by filtering for responses with fewer than five tokens in a neural setting. To reduce memory usage, we applied the low-rank adaptation scheme PEFT[6], adapted for 4 epochs.

In order to encourage a varied set of generations, we initially tried applying *diverse beam search* (Vijayakumar et al., 2016). However, this incurred a large latency cost – $O(N)$, where N represents the number of beams. Under *exact beam search*, each beam proved to be too similar, resulting in a narrow set of generations. Eventually, we resorted to sampling 20 examples with high temperature and low length, then taking the top-k most common responses – a memory-expensive but effective solution. In our experiments, we use $k = 3$ and always add "yes" to the set of candidates (as it is the most common).

For high-quality response generation, we used a Falcon-40B[7] model. We deploy this model on 4 GPUs, using an optimized Rust-based implementation with Flash-Attention for latency. In addition, we use Celery[8] to perform asynchronous processing. At prediction time, we send a HTTP request whose result is ignored; therefore, additional latency at end-of-turn is 0. On average, our Celery latency is between 1 to 3 seconds. We only send a prefiring request during neural-to-neural turns (i.e. last turn was neural and the next turn will also be neural). If the last turn was symbolic (i.e. written in CAMEL), we assume that Goldfinch should be sufficient.

On every turn, we attempt to retrieve prefiring responses from the prefiring server. If we do not find a prefiring response, we continue as normal, with an associated latency cost of around 5 to 10ms. However, if we do find a prefiring response, we use the response if the proposed response is "yes" or "no" and the observed response matches our Yes/No regexes respectively, or if there is an exact string match between the observed and user response.

On our test set of low-initiative utterances, a baseline of {"yes", "yeah"} achieves a hit rate of 15%. Our fine-tuned model, sampled at top-3, achieves a hit rate of 18%. Finally, augmenting the fine-tuned model with {"yes", "yeah"} achieves a hit rate of 25% (Figure 8).

Qualitatively, as the user reduces in initiative, we have a better chance of prefiring successfully, and we correspondingly take over more initiative (the larger 40B model is more loquacious than our default 3B model). In other words, prefiring becomes active precisely when needed – and automatically keeps the conversation alive.

[6] https://github.com/huggingface/peft
[7] https://huggingface.co/tiiuae/falcon-40b
[8] https://docs.celeryq.dev

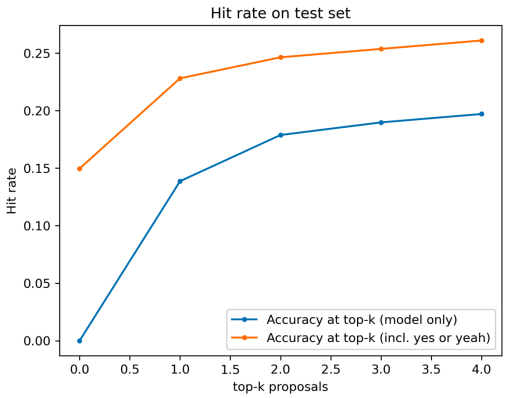


Figure 8: Hit rate against top-k proposals.

5 CAMEL

In order to facilitate flexible node-to-node transitions, we need to be able to composite nodes in arbitrary order. However, any component of a conversation has both implicit assumptions and side effects; executing these components in unexpected orders will lead to unusual behaviors of the dialog system, including crashes (as repeatedly happened in both Chi et al. (2021) and Paranjape et al. (2020)). This lack of guarantees fundamentally stems from the *imperative* nature of executable dialogue nodes (e.g. written in Python), which, although appealing for their flexibility, are hard to interpret.

To this end, we decided to scrap our dialog nodes from Paranjape et al. (2020), replacing them with an entirely new set of conversational responses written in a declarative language. We call this language **CAMEL** (**C**onversation-**A**ware **M**arkup **E**diting **L**anguage), which has the following advantages:

- **A consistent interface:** Accessing neural APIs, external databases, conjugation and pluralization, and variable storage is all handled through a consistent interface. This allows not only for significantly faster dialog writing, but also for an *error boundary*, *i.e.* the surrounding code is automatically resilient.[9]
- This consistency also allows for dialogues to be **automatically generated** using large language models, following advances in code generation (Chen et al., 2021).
- Finally, it is easy to write; members of our team with no previous coding experience were able to create CAMEL nodes without difficulty.

CAMEL enables the following abilities: (1) the ability to simply specify a dialogue node; (2) the ability to substitute context into conversational responses, including external API call results and more advanced, non-stateful neural models; (3) the ability to record user information and utilize it at a later time. Additionally, it significantly reduces the barriers to building chatbots; very little explicit Python is required, and it's easy to write simple extensions that integrate additional features into the language. Finally, CAMEL files are easily interpretable. We demonstrate these advantages via the example CAMEL syntax in our Appendix G.

Our language closely follows the information-state (IS) based approach to dialogue management (Traum and Larsson, 2003) in the following ways. First, the CAMEL language allows for state updates and state-informed decision-making, a key component of IS. These features remove dependencies on pre-specified dialogue structures, in contrast to the structure-based approach for dialogue modeling (Traum and Larsson, 2003). Additionally, the philosophy that different CAMEL files are constructed in isolation to others removes many limits on the number of available dialogue moves (actions such as which response Chirpy selects), further distinguishing CAMEL from structure-based approaches. Lastly, though CAMEL files are modeled separately, Chirpy incorporates the user's intentions into its dialogue management system, providing slight alignment with plan-based dialogue modeling (Cohen,

[9]This is similar to the error boundary concept in modern programming languages.

Figure 9: Engagement metrics vs. rating.

2019). Chirpy responds to navigational intent and each CAMEL file can define its own takeover conditions, adjusting the bot's response in the presence of unexpected user actions.

We parse CAMEL using Lark, a Python-based CFG parser, treating it as a compiled language by storing raw bytecode and only recompiling on file changes.[10]

5.1 NLG language

In order to generate text, we provide a flexible, template-like NLG language inspired by Inform 7 (Reed, 2010). **Databases** abstract lookups to external information sources; the `@lookup` NLG component can be used to search for a key, which is typically an entity or a string. Currently, we use this to look up data as diverse as news articles, ESPN basketball results, and entity-related factoids. **Neural generation** (`neural_generation`) allows for the interpolation of a neural response from a BlenderBot-3B model, either with a prompt or only given the conversational history. To speed up our neural model inference, we cache layer outputs for previously seen user and bot utterances as key-value pairs to reduce the amount of processing needed for new inputs. One can also **inflect** words by pluralizing verbs, nouns, and articles, and call arbitrary **Python** functions using the `nlg_helper` component.

5.2 Utilities for writing CAMEL

CAMEL Linter: We created a CAMEL linter (Hynes et al., 2017) which automatically identifies bugs at compilation time. This helps catch bugs before the bot is deployed, preventing crashes. In particular, the linter validates that our CAMEL code returns strings in node responses and does not attempt to reference `None`-type variables. At one point, the linter caught 240 errors – saving countless hours of developer work.

Language plugin: We released a plugin for VSCode, TextMate, and PyCharm (editors used by our team) which supports CAMEL. We plan to open-source all of this architecture once the competition concludes. An example of linter outputs is in Figure 16 in Appendix.

6 Supernode Overview

We have 2859 nodes, of which 2757 are specific. The 20 most frequently visited supernodes are displayed in Figure 10, which visualizes data collected between March 8 and June 24. Figure 9 shows engagement metrics plotted against rating over the same time period.

6.1 SPECIFIC nodes

Towards our goal of broad entity coverage, we create a large number of nodes using Goldfinch (Section 3) pertaining to various entities. We have nodes for top entities from various categories (e.g., people, places, and foods). To select these entities, we select the top k entities (500-1000 per category) brought up by humans. This strategy is effective, responding to 78% of utterances containing entities during `FOOD__intro` and 17% amongst all supernodes from July and August's

[10] https://lark-parser.readthedocs.io/en/latest/

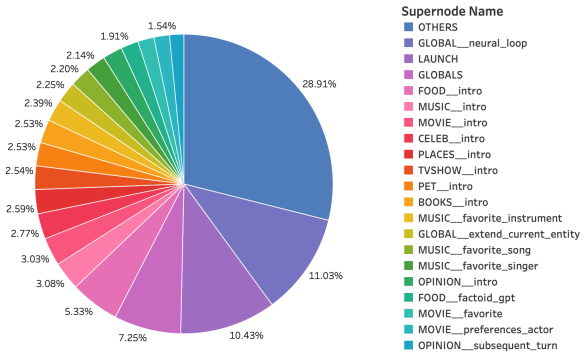


Figure 10: Top-20 Most Visited Supernodes from March to June. This figure shows each supernode's traffic percentage. "OTHERS" represents the frequency of the remaining 2839 supernodes.

rated conversations. Each specific node defines 1 turn (on average) of neural follow-up; some nodes have been manually deepened, notably the `SPECIFIC__HarryPotter` node, due to high volume.

6.2 Utility nodes

These run for large categories of entities and are broadly applicable to a variety of scenarios.

FACTOID Factoids for entities in our various databases are GPT-generated. For instance, apart from discussing the user's favorite food, the Food supernode is equipped to provide factoids whenever a user mentions food entities that are present in the database. We also have factoids related to entities such as books and celebrities. To ensure the factuality of our factoids, we enlisted manual verification of crowd workers.

GLOBAL_EXTEND_ENTITY The bot avoids frequent topic switches through this node, which extends the conversation about the current entity, if one exists, with a neural model; this model is prompted with information about the current entity.

NEURAL FALLBACKS In the case that the user provides an unexpected response, we fall back on a neurally generated response.

6.3 Entity-driven Specializations

Here, we describe *topic specializations*: a set of nodes that pertain to a particular topic, such as food, movies, or TV shows.

Intro nodes These initiate a conversation about a topic. Usually, we start by asking the user for their favorite X (for some topic X).

Food Who doesn't love talking about their favorite foods? The **<FOOD_*>** supernodes enable the bot to carry on in-depth conversations about food – ranging from providing insightful food factoids to asking the user for their favorite gastronomical memory. We handle each of the 1,000 most common foods that that users have ever brought up – plus a large number of foods scraped from Wikipedia.

Celebrity People often have celebrities that they follow and are more than happy to talk about them. To handle celebrity-related entities, we use a series of **<SPECIFIC_(person)>** nodes generated via Goldfinch (Section 3) which remark upon the celebrity's greatest achievements and interesting facts.[11] **<CELEB_discuss_*>** asks about the user's favorite movie / TV show / song by the celebrity, then

[11]For example, **<SPECIFIC_Bryan_Cranston>** has the following prompt: "Did you know that Bryan Cranston played both Hal in Malcolm in the Middle and Walter White in Breaking Bad? Which one of his characters do you prefer?"

queries about their opinion. <CELEB_generic> involves general discussions around the celebrity that could be applied to unknown celebrities as well[12] to maintain conversation flow.

Movie and TV Show To handle movie-related entities, we provide several <MOVIE_*> and <TVSHOW_*> supernodes, which engage with the user about their favorite movies and TV shows, respectively. To create a set of movie entities, we used IMDB, generating concise one-sentence plots and thoughts about each movie using GPT-3 (Brown et al., 2020). <MOVIE_favorite> begins movie-related discussions by asking about the user's favorite movie, and <MOVIE_aspect> drills deeper by asking about specific components of interest of that movie. <MOVIE_preferences_actor> transitions the topic to actors involved in that movie. <TVSHOW_qa> asks point-in-time relevant, ChatGPT-generated questions for individual episodes, carefully designed to avoid spoiling the show.[13]

Books In order to handle books, <BOOKS_intro> queries the user's favorite book, then we discuss the author of this book by providing a factoid in <BOOKS_author> . The bot then asks whether the genre of this book is the user's genre of choice in general (<BOOKS_genre>). If the user responds positively, we then proceed to a genre-based recommendation. The recommendation is pre-selected either by random selection for smaller genres or via pre-cached summary-based semantic clustering using Sentence Transformers (Reimers and Gurevych, 2019).

6.4 Database-driven Specializations

These sets of nodes typically handle less structured data that don't fit nicely into Wikipedia entities.

Music The Music specialization enables the bot to engage with users about music. It prompts users with questions about their music preferences, covering genres, songs, singers, instruments, and compositions. This specialization is more database-driven than the aforementioned ones. Each of these aspects of music corresponds to multiple databases. Additionally, most responses have been pre-generated using ChatGPT, as opposed to the often well-defined, on-the-fly slot-filling approach employed in other specializations. Entity identification for this specialization has been fortified using specific database constructions. Both song names and artist names are included in the song database, and music groups are stored along with their members as well.

Sports The <SPORTS__*> nodes focus on providing up-to-date conversations about sports; currently, we focus on the NBA 2022-2023 season. To ensure real-time information for each game in the semifinals, conference finals, and season finals, we scrape the ESPN API, which is utilized to generate two noteworthy events summarizing each game for the respective teams using ChatGPT.

6.5 Open-Ended Specializations

Pets The average pet owner is proud of their pet and would like to discuss them with Chirpy Cardinal. The 8 <PETS__*> nodes first ask about the user's pet, further inquiring about the type of animal and breed. We generated a pet-related ontology that allows us to infer, for example, that if the user has a golden retriever that they also have a dog. Notably, if the user explicitly says that they lost their pet, the bot will avoid talking about their pet directly and acknowledge their loss instead.

Personal Issues Many users—especially those who chat with our socialbot looking for companionship—share personal struggles with our bot, requiring emotional sensitivity and tact. Handling such conversations purely neurally would result in rapid degeneration due to neural toxicity Dinan et al. (2021). To address this, the PERSONAL ISSUES nodes respond to personal disclosures using active listening techniques (Bodie et al., 2015), asking exploratory questions about the nature of the user's issue (*"When did you start feeling this way?"*), and validating their concerns (*"I see, that sounds difficult"*). We provide the conversation flow in Figure 11.

Chirpy activates the personal issues supernode when it detects a negative-sentiment utterance that matches our comprehensive personal issues regex; this regex searches for common phrases about

[12]For example, "What do you think makes Bryan Cranston so successful?"

[13]For example, a question for *Breaking Bad*'s Season 1 Episode 4 is: "Why do you think Jesse tried to make amends with his parents after Walt told his family about his cancer?"

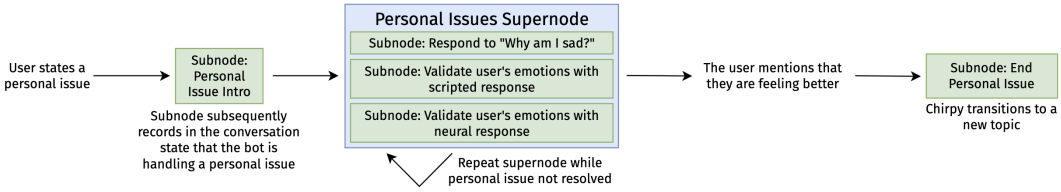


Figure 11: The conversation flow inside of the personal issues supernode.

afflictions, negative emotions (for example, anger and frustration), and relationship issues. After detecting a personal issue, Chirpy thanks the user for sharing, acknowledges that their situation is unfortunate, and offers to listen (Dean and Jr., 2013).

Aliens The Aliens specialization provides a thoughtful soliloquy on the subject of extraterrestrial life, ranging from the possibility of alien civilizations to the enigmatic "Oumuamua," a mysterious asteroid found in 2017. Note that this specialization only occurs once the user has chatted for 30 turns or more (the exception being if the user provided navigational intent to a space-related topic).

This node is notable because it suggests higher-order thought beyond the confines of the Echo Show. Over the final two weeks of the SGC5 Semifinals, the average rating for these conversations was 3.77 $\pm$ 1.33. 65% of these conversations spent at least three turns discussing aliens, showing high user engagement with this topic.

7 Multimodal dialogue

7.1 Midjourney

Visual engagement is a powerful component of what makes conversations interesting. Thus, we decided to accompany Chirpy's statements with watercolor images generated through Midjourney.[14] In addition to ensuring consistency in image style and quality, using colorful art with the hand-drawn element specific to watercolor gives our bot a human touch. This human element assists Chirpy in creating a connection with the user and ultimately improves the conversation experience.

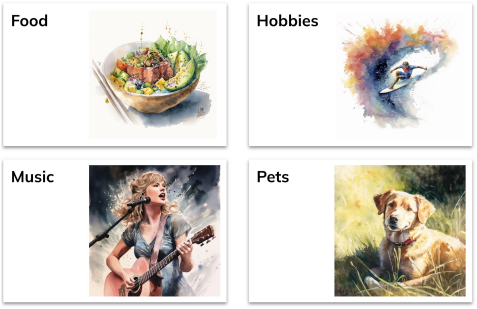
To generate images that are realistic and not too different from what users expect, we performed some basic prompt engineering. For example, through observation, we identified that adding "in the sunlight" to food prompts generated aesthetic and accurate images most consistently. See Figure 12.

Figure 12: Prompt engineering examples.

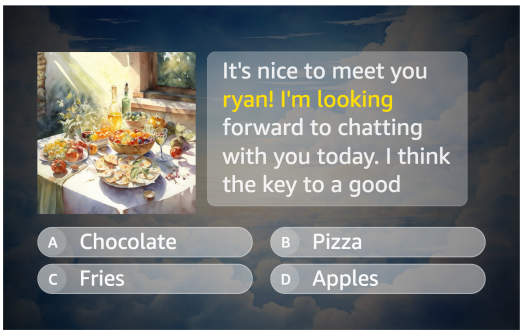
7.2 Suggestions, Karaoke, and Avatar

For many of our supernodes, we record the most common user responses and add them as **suggestions**, using the Alexa Presentation Quiz template (Figure 13a). This provides a speechless option for interacting with our chatbot. Moreover, during each utterance, we use Alexa Presentation Karaoke Mode to highlight the line that is being spoken, for a more interactive experience (Figure 13b). Finally,

[14] https://www.midjourney.com



(a) Images generated per supernode.



(b) Karaoke mode and suggestions.

Figure 13: Additional multimodal dialogue artifacts.

we don't have a specific image to use, humanistic person & bot images provide a sense of AI-human camaraderie and increase trust.

8 Discussion and Future Work

In this white paper, we have proposed a framework for rapidly building and distilling LLMs into customizable, easily interpretable chatbots. We present a framework that is systematic while remaining flexible, expressive while remaining interpretable – all with the expressly stated goal of compelling, cogent conversation. That said, we do not claim that our principles and techniques are sufficient, or exclusively necessary; we hope they can serve as a guiding point for further chatbot development.

Applicability The approaches described in this white paper have wide applicability, beyond just socialbots. The **CAMEL** system serves as an easily extensible platform for building dialogue agents that allows for the easy composition of neural generation and handwritten dialogue. In particular, we believe that CAMEL might even be more appropriate for *taskbots* or other goal-oriented dialogue, where one might want to insert specific manually written lines of dialogue that advance a goal, while retaining neural flexibility. As a general principle, **Goldfinch**—distilling neural models into a dialogue tree—can be applied to any such tree-based system to enormously apply its flexibility. Finally, we believe that **Kingfisher** is the most widely applicable of all: in any chat-based application in which one can relatively easily predict the next user utterance, Kingfisher can be helpful in pre-caching computation costs.

Neural Prediction: Goldfinch/Kingfisher Synergy In this work, we've presented two major methods for approximating the fluency and power of neural systems: **Goldfinch**, where a large number of user responses are enumerated for each possible "dialogue situation" (as expressed as a conversational tree node), and **Kingfisher**, where a smaller number of user responses are enumerated on-the-fly. Notably, these two approaches express a similar idea, just orthogonally: Goldfinch covers cases in which the dialogue situation is known (e.g., if we've recently asked a static question), and Kingfisher can handle cases we may not have seen in advance (e.g., during a long string of neural generation).

Speed/quality tradeoff We encountered a **speed vs. quality tradeoff** when designing our dialogue. Using a large amount of hand-written dialogue results in an extremely *precise* but *jerky* experience: the conversation always correctly pertains to what we are saying, and we have very high initiative, but the conversation moves quickly without regard to listening to the user. On the other hand, using a large amount of neural is *slow*: we correctly address the user's statements but have very low initiative, so the conversation seems to be going nowhere.

Social Impact In this work, we have presented a conversational agent that conducts an open-domain dialogue. We believe that many people would enjoy having a chat partner who is empathetic and knowledgeable, and our ratings seem to suggest that a reasonable number of people appreciate their conversations enough to want to talk to the bot again. That said, the use of neural generative

models and open-domain-retrieved text means that there does exist a risk that users may be exposed to unsafe utterances or discussion topics. Conversational models of all kinds can produce sexist, racist, or otherwise unsafe statements; neural conversational agents can be particularly vulnerable due to pre-training on Internet chat forums, which can be particularly toxic (Xu et al. (2020); Dinan et al. (2021)). Finally, the human-like nature of open-domain dialogue systems can be particularly damaging when used in an adversarial context, e.g., by state actors Boshmaf et al. (2012). Ultimately, like all text generation methods, the benefit of releasing an open-domain dialogue model must be weighed against its possible downsides.

References

James F Allen, Bradford W Miller, Eric K Ringger, and Teresa Sikorski. 1996. A robust system for natural spoken dialogue. *arXiv preprint cmp-lg/9606023*.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. Falcon-40B: an open large language model with state-of-the-art performance.

Graham D Bodie, Andrea J Vickery, Kaitlin Cannava, and Susanne M Jones. 2015. The role of "active listening" in informal helping conversations: Impact on perceptions of listener helpfulness, sensitivity, and supportiveness and discloser emotional improvement. *Western Journal of Communication*, 79(2):151–173.

Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. 2012. Key challenges in defending against malicious socialbots. In *5th {USENIX} Workshop on Large-Scale Exploits and Emergent Threats ({LEET} 12)*.

Samuel Broscheit. 2019. Investigating entity knowledge in BERT with simple neural end-to-end entity linking. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 677–685, Hong Kong, China. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Davide Castelvecchi. 2016. Can we open the black box of AI? *Nature News*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ethan A. Chi, Ashwin Paranjape, Abigail See, Caleb Chiam, Trenton Chang, Trenton Kenealy, Swee Kiat Lim, Amelia Hardy, Chetanya Rastogi, Haojun Li, Alexander Iyabor, Yutong He, Hari Sowrirajan, Peng Qi, Kaushik Ram Sadagopan, Nguyet Minh Phu, Dilara Soylu, Jillian Tang, Avanika Narayan, Giovanni Campagna, and Christopher D. Manning. 2022. Neural generation meets real people: Building a social, informative open-domain dialogue agent. In *Proceedings of the 23rd Annual SIGdial Meeting on Discourse and Dialogue*, Edinburgh, Scotland. Association for Computational Linguistics.

Ethan A Chi, Chetanya Rastogi, Alexander Iyabor, Hari Sowrirajan, Avanika Narayan, and Ashwin Paranjape. 2021. Neural, neural everywhere: Controlled generation meets scaffolded, structured dialogue. *Alexa Prize Proceedings*.

Cheng-Han Chiang and Hung yi Lee. 2023. Can large language models be an alternative to human evaluations? *ACL*.

Philip R. Cohen. 2019. Foundations of collaborative task-oriented dialogue: What's in a slot? *ACL*.

Marleah Dean and Richard L. Street Jr. 2013. A 3-stage model of patient-centered communication for addressing cancer patients' emotional distress. *Patient Education and Counseling*.

Ameet Deshpande, Vishvak Murahari, Tanmay Rajpurohit, Ashwin Kalyan, and Karthik Narasimhan. 2023. Toxicity in ChatGPT: Analyzing persona-assigned language models. *arXiv preprint arXiv:2304.05335*.

Jean-Louis Dessalles. 2017. Conversational topic connectedness predicted by simplicity theory. In *39th Annual Conference of the Cognitive Science Society*, pages 1914–1919.

Jwala Dhamala, Tony Sun, Varun Kumar, Satyapriya Krishna, Yada Pruksachatkun, Kai-Wei Chang, and Rahul Gupta. 2021. Bold: Dataset and metrics for measuring biases in open-ended language generation. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, page 862–872, New York, NY, USA. Association for Computing Machinery.

Emily Dinan, Gavin Abercrombie, A Stevie Bergman, Shannon Spruit, Dirk Hovy, Y-Lan Boureau, and Verena Rieser. 2021. Anticipating safety issues in e2e conversational ai: Framework and tooling. *arXiv preprint arXiv:2107.03451*.

Delbert Dueck. 2009. *Affinity propagation: clustering data by passing messages*. University of Toronto Toronto, ON, Canada.

Yanai Elazar, Nora Kassner, Shauli Ravfogel, Abhilasha Ravichander, Eduard Hovy, Hinrich Schütze, and Yoav Goldberg. 2021. Measuring and improving consistency in pretrained language models. *Transactions of the Association for Computational Linguistics*, 9:1012–1031.

Boris Galitsky and Dmitry Ilvovsky. 2017. Chatbot with a discourse structure-driven dialogue management. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 87–90.

Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A. Smith. 2020. RealToxicityPrompts: Evaluating neural toxic degeneration in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3356–3369, Online. Association for Computational Linguistics.

Herbert P Grice. 1975. Logic and conversation. In *Speech acts*, pages 41–58. Brill.

Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. 2023. The false promise of imitating proprietary llms.

Eric J. Horvitz. 1999. Principles of mixed-initiative user interfaces. In *CHI '99: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 159–166.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Nick Hynes, D Sculley, and Michael Terry. 2017. The data linter: Lightweight, automated sanity checking for ml data sets. In *NIPS MLSys Workshop*, volume 1, page 5.

Michael Johnston, Cris Flagg, Anna Gottardi, Sattvik Sahai, Yao Lu, Samyuth Sagi, Luke Dai, Prasoon Goyal, Behnam Hedayatnia, Lucy Hu, Di Jin, Patrick Lange, Shaohua Liu, Sijia Liu, Daniel Pressel, Hangjie Shi, Zhejia Yang, Chao Zhang, Desheng Zhang, Leslie Ball, Kate Bland, Shui Hu, Osman Ipek, James Jeun, Heather Rocker, Lavina Vaz, Akshaya Iyengar, Yang Liu, Arindam Mandal, Dilek Hakkani-Tür, and Reza Ghanadan. 2023. Advancing open domain dialog: The fifth alexa prize socialbot grand challenge. In *Alexa Prize SocialBot Grand Challenge 5 Proceedings*.

Tuan Manh Lai, Quan Hung Tran, Trung Bui, and Daisuke Kihara. 2020. A simple but effective bert model for dialog state tracking on resource-limited systems. *ICASSP*.

Lenin Medeiros, Charlotte Gerritsen, and Tibor Bosse. 2019. Towards humanlike chatbots helping users cope with stressful situations. In *Computational Collective Intelligence: 11th International Conference, ICCCI 2019, Hendaye, France, September 4–6, 2019, Proceedings, Part I 11*, pages 232–243. Springer.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2023. Factscore: Fine-grained atomic evaluation of factual precision in long form text generation. *arXiv preprint arXiv:2305.14251*.

OpenAI. 2023. Gpt-4 technical report.

Ashwin Paranjape, Abigail See, Kathleen Kenealy, Haojun Li, Amelia Hardy, Peng Qi, Kaushik Ram Sadagopan, Nguyet Minh Phu, Dilara Soylu, and Christopher D Manning. 2020. Neural generation meets real people: Towards emotionally engaging mixed-initiative conversations. *arXiv preprint arXiv:2008.12348*.

Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language models as knowledge bases? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2463–2473.

Aaron Reed. 2010. *Creating interactive fiction with Inform 7*. Course Technology Press.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks.

Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Eric Michael Smith, Y-Lan Boureau, and Jason Weston. 2021. Recipes for building an open-domain chatbot. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 300–325, Online. Association for Computational Linguistics.

Yiqiu Shen, Laura Heacock, Jonathan Elias, Keith D Hentel, Beatriu Reig, George Shih, and Linda Moy. 2023. Chatgpt and other large language models are double-edged swords.

Svetlana Stoyanchev, Simon Keizer, and Rama Doddipatla. 2021. Action state update approach to dialogue management. *ICASSP*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

David R. Traum and Staffan Larsson. 2003. The information state approach to dialogue management. *Current and New Directions in Discourse and Dialogue*.

Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. 2016. Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*.

Nick Webb. 2000. Rule-based dialogue management systems. In *Proceedings of the 3rd International Workshop on Human-Computer Conversation, Bellagio, Italy*, pages 3–5.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Joseph Weizenbaum. 1966. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45.

Jing Xu, Da Ju, Margaret Li, Y-Lan Boureau, Jason Weston, and Emily Dinan. 2020. Recipes for safety in open-domain chatbots. *arXiv preprint arXiv:2010.07079*.

Yunzhi Yao, Peng Wang, Bozhong Tian, Siyuan Cheng, Zhoubo Li, Shumin Deng, Huajun Chen, and Ningyu Zhang. 2023. Editing large language models: Problems, methods, and opportunities. *arXiv preprint arXiv:2305.13172*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*.

A Acknowledgments

Thank you to Ethan A. Chi for invariably helpful discussions and advice throughout the course of the entire year. We would also like to acknowledge the entire SGC5 Amazon team: Cris Flagg, Sattvik Sahai, Reza Ghanadan, Anna Gottardi, Yao Lu, Yang Liu, Heather Rocker, Akshaya Iyengar, Chao Zhang, Daniel Pressel, Desheng Zhang, Hangjie Shi, James Jeun, @jbrockeo, Lavina Vaz, Leslie Ball, Lucy Hu, Luke Dai, Michael Johnston, Osman Ipek, Prasoon Goyal, Samyuth Sagi, Shaohua Liu, and others. Finally, we would like to thank John Hewitt, Eric Zelikman, Patrick Liu, Jack Xiao, Haojun Li, Rylan Yang, Moussa Doumbouya, and Erica Chi for constructive feedback and contributions.

B State implementation

State is implemented as a key-value store which is shared by all supernodes; however, only the responding supernode ever writes to state. We save the state object to a PostgreSQL database, allowing us to identify returning users and continue our conversation from where we ended.

C Additional Details on NLU

Node designers have read and write access to a list of flags that will be populated during each turn, and they can design nodes with these flags without studying their underlying implementation.

The majority of our NLU matches regexes and searches for keywords, among other operations. While previous works have proposed neural-based slot filling (Lai et al., 2020; Stoyanchev et al., 2021), these models are limited to specific subject matters and require extensive labeling efforts to develop. Such resources are difficult to curate for an open-domain chatbot.

We split NLU into four categories, as enumerated and described below.

C.0.1 Global NLU

The bot runs the `global_nlu.py` file every turn. Global NLU provides globally applicable information about the turn, such as whether the utterance is a yes/no statement.

C.0.2 Nonlocal NLU

Nonlocal NLU lives in each supernode's `nlu.py` file. It also provides globally applicable NLU, but we split topic-specific NLU into their respective supernodes to prevent the global NLU file from growing too large. For example, Chirpy has a supernode about common activities; in this supernode's nonlocal NLU, Chirpy detects whether an utterance mentioned any activities, helpful information to have. A supernode's nonlocal NLU is only executed while the bot uses a different supernode to generate the follow-up response.

C.0.3 Local NLU

Both local and non-local NLU are defined in each supernode's `nlu.py` file. Once the bot selects a supernode to provide the follow-up response, it runs that supernode's local NLU. Local NLU provides turn- and topic-specific NLU. For example, if the bot is responding to someone saying their name, then the local NLU will contain code that detects this name.

C.0.4 Clarification NLU

Supernodes can optionally define clarification NLU. This NLU detects a user's attempt to clarify their previous utterance and extracts the information necessary to provide an accurate response. If a clarification attempt is detected, the previous turn's follow-up supernode will run again, with the clarified information in mind. Chirpy uses clarification NLU in the Pets supernode to allow users to correct whether or not they own a pet if they misspoke.

D Supernode weighting

Nodes are weighted in the following order, with each level being approximately 0.5-1 order of magnitude above the next in the weighted random distribution:

1. Nodes that respond to a *flag* have the highest ranking. This only occurs rarely, i.e. when the user has taken on an unusual amount of initiative; or the bot has labeled the situation as being special in some way.
2. Nodes that handle the current entity—*and only that current entity*.
3. Nodes that use the current entity in some way, such as the **<Factoid>** node.

E Conversational Graph

This visualization shows our system's conversational graph. An edge between two supernodes indicates that they were activated in consecutive dialogue turns.

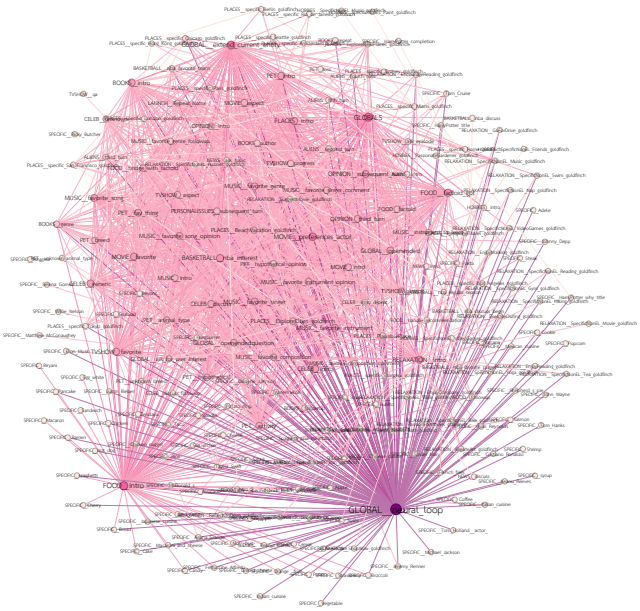


Figure 14: Chirpy Cardinal 3.0's conversational graph (edge color demarcates frequency of traversal). Note the extreme interconnectedness, which was not possible in any of our previous incarnations.

F Naive Bayes

More specifically, let us represent sentences as vectors in $\mathbb{R}^d$. Assume the scenarios are mutually exclusive. Then, by Bayes, we have the log-likelihood of being in some cluster c as:

$$\begin{aligned}\log[p(y=c|x)] &= \log\left[\frac{p(x|y=c)p(c)}{\Sigma_{d\in C}p(x|y=d)p(d)}\right] \\ &\sim \Sigma_{j=1}^{d}\log[p(x_j|y=C)]\log[p(c)]\end{aligned}$$

Let us defer the problem of estimating $p(c)$:

$$\begin{aligned}&\sim \Sigma_{j=1}^{d}\log[p(x_j|y=C)] \\ &\sim \Sigma_{j=1}^{d}\log\left[\frac{\text{number of sentences in cluster C that contain word } j}{|C|}\right]\end{aligned}$$

Applying Laplace smoothing to avoid 0 probabilities / neginf log-likelihoods, we obtain

$$\sim \Sigma_{j=1}^{d} \log \left[\frac{1 + \text{number of sentences in cluster C that contain word } j}{|C| + d} \right]$$

G Sample CAMEL File

Example: `FOOD__intro` node

```
<ENTRY_CONDITIONS>
IS_NONE(State.favorite_food) or IS_TRUE(Flags.FOOD__mentioned)

<PROMPTS>
Prompt favorite_food [ ] {
    "What's one of your favorite things to eat?"
}

<SET_STATE>
State.favorite_food = @Utilities.current_topic # (1) Variables

<SUBNODES>
Subnode make_ingredient_attribute_comment [
  EXISTS(database_name="food_data", key={State.favorite_food.name | lower}, "
      ingredient")
] {
    - "Ah yes,"
    - @Utilities.current_topic
    - @inflect(inflect_token="is", inflect_entity=Utilities.current_topic) #
        (2) Contextual inflection
    - "one of my favorite things to eat up here in the cloud."
    - "I especially like its use of"
    - @lookup(database_name="food_data", key={State.favorite_food.name |
        lower}, column="ingredient") # (3) Database lookup
}

Subnode food_intro_no [ IS_TRUE(Flags.GlobalFlag__NO) ] { # Entry condition
    in brackets
   - "Ok! Let me know if you ever have a recommendation!"
}

Subnode food_intro_default [ ] {
  - @neural_generation(prefix={ # (4) Conditional neural generation
     - "I especially love how"
  })
}
```

Figure 15: An example of CAMEL syntax. Entry conditions for subnodes are surrounded in brackets.

H CAMEL Linter Specifics

We check for these properties in the following manner:

1. Take a supernode S.
2. Validate its predicates. We confirm that each verb exists in the CAMEL language; CAMEL cannot compile if someone uses a non-existent verb because the compiler cannot map that verb to a Python object. We also confirm that the predicates do not access attributes on `None`-type variables.

3. Log variable protection. A variable is protected if its value is checked against `NOT IS_NONE`, `IS_INSTANCE`, or `EXISTS` verbs before its usage; if these verbs return `True`, then the variable will not be of `None`-type, so it can be safely referenced within nodes.
4. Log variable types. The linter analyzes each node's state-update portion, and infers variable types based on what each variable is assigned to. The linter infers variable v's type by recursively inferring the type that v is assigned to. Eventually, the recursion stumbles upon a value whose type is defined in the CAMEL specification or by a type annotation in CAMEL's underlying Python code.
5. Validate the response NLGs. Specifically, we check that the response NLGs only concatenate strings when generating the response and do not access attributes on `None`-type variables. Steps 1-4 make this step straightforward.

Once we perform these steps on all supernodes, we will have solved many errors in our CAMEL code.

The linter is used as a command-line tool. Since it outputs a lot of information, we use the Colorama library to color-code linter outputs, vastly improving readability. Figure 16 contains a sample output from the CAMEL linter.

Here is some additional information on how we perform the above steps.

1. Validating predicates: we keep a list of implemented verbs (IS_TRUE, IS_NONE, etc.) and confirm that each predicate only uses verbs from this list. The linter also checks that the predicate does not access attributes on variables that are None-type.
2. Logging protection: this logging is one of the most important parts of this linter. Whenever the linter parses a predicate, it recursively searches the predicate for NOT IS_NONE, IS_INSTANCE, and EXISTS verbs. If a variable is wrapped in these verbs, it can be safely used in all parts of Camel code that execute when the predicate in question evaluates as true. For instance, if a subnode has an entry condition that checks that a variable v is not None, then we can use v inside of the subnode without risk of Chirpy crashing because we know that v is not None-type in that portion of Camel code. Since the variable is not None, we can safely perform actions such as attribute-accesses.
3. Logging variable types: all variables are assigned to the output of a NLG. We log variable types by inferring the output of a NLG. First, certain NLGs always return a specific type. For example, `@neural_generation` always returns a string. Thus, when a variable is assigned to the output of `@neural_generation`, we know that the variable's value is a string. Second, some NLGs return the value of another variable v. If we know v's return type, we know the current variable's type. Alternatively, if we know that v is an instance of a particular Python class, we can use Python type-annotations to infer the current variable's type.
4. Validating response NLGs: once we know which variables are protected and their return types, validating a response NLG is quite straightforward. We just need to ensure that each value the NLG concatenates to form a final response is a String, information that we now have.

```
↪ Parsing supernode TVSHOW__aspect
 entry_conditions
• Error: We cannot confirm that State.favorite_tvshow.qa_answered's property access is valid. Please check manually
• Error: We cannot confirm that State.favorite_tvshow.cur_season's property access is valid. Please check manually
• Error: We cannot confirm that State.favorite_tvshow.cur_episode's property access is valid. Please check manually
 PROMPT: ask_user_reason_if_finished:entry_conditions
• Error: We cannot confirm that State.favorite_tvshow.finished's property access is valid. Please check manually
 PROMPT: ask_user_reason_if_not_finished:entry_conditions
• Error: We cannot confirm that State.favorite_tvshow.finished's property access is valid. Please check manually
 SUBNODE: default:assignments
• Error: We cannot confirm that State.favorite_tvshow.qa_answered's property access is valid. Please check manually
↪ Parsing supernode OPINION__subsequent_turn
PROMPT: positive response:response_nlg
• Error: We cannot confirm that State.User__CurrentOpinionTopic.talkable_name's property access is valid. Please
      check manually
 locals
• Error: We cannot confirm that State.CurrentTopic.name's property access is valid. Please check manually
 SUBNODE: positive_response:response_n1g
• Error: We cannot confirm that State.CurrentTopic.name's property access is valid. Please check manually
• Error: NLGHelper(name='get_opinion_response', args="positive", Val(variable=Variable(namespace='State', name='
      CurrentTopic', subvariables=[Property(name=Token('nlg__PARAM_ARG_NAME', 'name'))]), operations=[])]) might
      not return a string (even if it is not none). Please confirm that it returns a string.
 SUBNODE: negative_response:response_n1g
• Error: We cannot confirm that State.CurrentTopic.name's property access is valid. Please check manually
• Error: NLGHelper(name='get_opinion_response', args="negative", Val(variable=Variable(namespace='State', name='
      CurrentTopic', subvariables=[Property(name=Token('nlg__PARAM_ARG_NAME', 'name'))]), operations=[])]) might
      not return a string (even if it is not none). Please confirm that it returns a string.
↪ Parsing supernode OPINION__third_turn
 PROMPT: new_topic:response_nlg
• Error: NLGHelper(name='get_topic_type', args=[Val(variable=Variable(namespace='State', name='CurrentTopic',
      subvariables=[Property(name=Token('nlg__PARAM_ARG_NAME', 'name'))]), operations=[])]) might not return a
      string (even if it is not none). Please confirm that it returns a string.
↪ Parsing supernode FOOD__factoid
 PROMPT: food_factoid:response_nlg
 ...
```

Figure 16: Sample linter output. Note that the linter output checks for errors such as accessing properties on unprotected variables and asks the developer to double check that a component of a response NLG is actually a string. Further note that color-coding makes these errors much easier to read.

I SPECIFIC Nodes

A sample Goldfinched SPECIFIC node that discusses the Harry Potter series (with slight modifications for optimal readability).

Example: `SPECIFIC__HarryPotter` node

```
<ENTRY_CONDITIONS>
IS_EQUAL(State.CurrentTopic.name, @constant("Harry Potter"))

<PROMPTS>
Prompt favorite_harry_potter_book_name [] {
    - "Which book in the Harry Potter series is your favorite so far?"
}

<SET_STATE>
State.SPECIFIC__HarryPotter_data.fav_part = Flags.
    SPECIFIC__HarryPotter_title_name

<SUBNODES>
Subnode SPECIFIC__HarryPotter_favorite_revealed [
    not IS_NONE(State.SPECIFIC__HarryPotter_data.fav_part)
] {
    # A favorite book was mentioned.
    - "That's actually my favorite as well!"
} <
    # These flags mark further points of discussion about Harry Potter.
    Flags.SPECIFIC__HarryPotter_discuss_book = @constant(True)
>

[
  Subnode SPECIFIC__HarryPotter_UserConfessesUnfamiliarity [
    IS_TRUE(Flags.SPECIFIC__HarryPotter_ConfessUnfamiliarity_goldfinch)
  ] {
      # The user is not very familiar with Harry Potter.
      - "That's ok! I'm sure you've heard about the magical adventures of
          Harry, Hermione, and Ron. It's definitely worth watching or reading
           it!"
  }

  Subnode SPECIFIC__HarryPotter_CurrentlyReading_goldfinch [
    IS_TRUE(Flags.SPECIFIC__HarryPotter_CurrentlyReading)
  ] {
      # The user is still finishing the series.
      - "No worries! I can understand why it's difficult to choose a favorite
          before finishing the entire series."
  } <
    Flags.SPECIFIC__HarryPotter_ask_for_fav_char = @constant(True)
  >
]
```

J Goldfinch Prompts

J.1 Generating Sample User Utterances

For each personality, we run the following prompt to generate ten unique user responses to a specific virtual assistant scenario:

> *We're trying to test a new Alexa chatbot.*
> *You are now a user that is talking to an Alexa chatbot. For context,* `$personality`*.*
> *Please provide 10 reasonable answers that normal users like yourself might respond with, separated by linebreaks.*
> *The answers should be specific and common, and about half of them should be reasonably complete statements that someone might say out loud.* `$context`*.*
>
> *User: My name is Emily*
> *Chatbot: It's nice to meet you Emily! I appreciate you taking the time to chat with me today. I was wondering, what's your favorite movie? Things you might say:*
> *1. I don't like movies*
> *2. I can't think of one*
> *3. i love Toy Story*
> *4. im a big fan of Frozen*
> *5. honestly i love the The Emoji Movie*
> *6. i do like Forrest Gump*
> *7. Jurassic Park*
> *8. how about Harry Potter*
> *9. Star Wars*
> *10. my favorite movie is Star Trek*

We showcase a few of the personality types we use as part of the prompt below:

> *1. you are a 30 year old that works in the relevant industry and is quite knowledgeable themself*
> *2. you do not have enough experience to answer the virtual assistant's prompt*
> *3. you are a hippy with a lot of unusual interests that might be less common*
> *4. you are a 70 year old that is lonely and wants to share any story from their life*
> *5. you are a 35 year old immigrant without a lot of money, and is just trying to make a living*
> *6. you are a 50 year old with three kids that you care a lot about*
> *7. you are a 30 year old living under very strict rules*
> *8. you are a rich 25 year old socialite and you are always looking for new possessions and experiences*
> *9. you are a 18 year old college student and you are living with a roommate*
> *10. you are a 10 year old kid who likes to watch television*

In order to ensure that our LLM outputs contain factual information, we leave space for optional context in the prompt, labelled as `$context`, that ensures that our outputs are grounded in factual information.

J.2 Generating Sample Virtual Assistant Responses

In order to simulate the desired behavior of our virtual assistant, we engineered a prompt to generate responses that aim to be not only personable, but also informative:

> *You are now an incredibly friendly, interesting chatbot that many people would love to have a conversation with.*
> *Your conversation is over audio, so we would like your response to be something that you can say out loud without sounding awkward.*
> *Also, you should have some interesting, dramatic flair that shows you're worldly*

and interesting and know something about the topic at hand. Be somewhat specific. You should not begin your statement with a question.

J.3 Deepening Prompt

Complete the bot utterance with a specific question that's particularly relevant to the topic at hand. (For example, if we were talking about golfing, ask if you have a preferred club or kind of swing.) Ask extremely unusual questions that are rather specific, preferably naming specific attributes of the topic at hand. Don't just ask "What's your favorite thing about it," as people will find that kind of boring. You should be exciting as possible, while being rather conversational!

J.4 News Dialogue Prompting

J.4.1 High-Initiative Users

Our setup for ChatGPT prompting is as follows:

1. We prepend the message of system specification {`"role": "system", "content": "You are a helpful and friendly assistant."`} when calling the API.
2. Retrieved news stories are appended to specific prompts to generate both the dialogue-like paraphrase and the socialbot's opinion on the news story.

The prompt used for generating dialogue-like paraphrases is:

`Rephrase this news summary to a natural, easy-to-understand, spoken conversation utterance, without any greeting:.`

For generating opinions on news pieces, we use the following prompt:

`Give a relatable, natural, one-sentence opinion on the following news story.`

J.4.2 Lower-Initiative Users

The prompt we use to generate next-turn bot responses is the following:

`You are now an incredibly friendly, interesting chatbot that many people would love to have a conversation with.`

`We would like your response to be something that you can say out loud without sounding awkward. It should be no more than 15 words long.`

`You are aware of the following information: <EVIDENCE>`

J.5 GPT-based evaluation

For PLACES, we generated 100 responses to the question "I'm hoping to plan a vacation soon and wanted to get some advice from you: what's a city that you enjoy visiting?" using GPT-3.5 and the following prompt:

You are now a user that is talking to an Alexa chatbot. You're a typical American consumer that talks to the chatbot like a normal person. The conversations are over audio, so don't be overly verbose. Don't generate more than 2 sentences per response, and try to talk normally. Under no conditions may you say more than 20 words. Rephrase your statement to use simpler language so it fits within 20 words. Feel free to mention some lesser known subjects.

Make 100 diverse responses to the question: "I'm hoping to plan a vacation soon and wanted to get some advice from you: what's a city that you enjoy visiting?" Remember you are in a casual conversation. Think about all of the possible types of responses a person could say to this question. Maybe they like to go abroad. Maybe they like to stay in America. Maybe they dislike traveling.

Maybe they have a difficult time choosing one city. These are just some examples. Make your 100 responses diverse and make sure they will make sense in a casual conversation. Try to cover every possible type of response you can think of with these 100 responses. Remember, your responses should be directly relevant to the original question and they should be diverse. Never repeat any idea in multiple responses.

We provided each of these responses to a Goldfinched and a handwritten version of PLACES. We asked GPT-3.5 to compare these conversations and select the better one using chain-of-thought-reasoning, sampling a reasoning at temperature 1 and then the final answer at temperature 0. We had GPT make this comparison 5 times for each conversation, and determined the majority vote as GPT's ultimate classification. Applying this method, we had GPT compare these 100 conversations 3 times for a total of 300 evaluations. In total, we observed 183 wins for Goldfinch-PLACES and 117 wins for Handwritten-PLACES.

We performed a similar experiment on RELAXATION, comparing sample conversations deepened by Goldfinched RELAXATION nodes to simply extending a conversation with neural turns. Across 36 pairwise-comparisons, GPT preferred Goldfinched relaxations 25 times to 11 for neural.

On our note on LLM-evaluation's consistency, GPT provided the same preference in each trial for 45 out of 100 PLACES conversations and 7 out of 12 RELAXATION conversations.

The GPT comparison prompt is as follows:

- System: *You are helping us evaluate a chatbot. You will be asked to classify certain statements. Do NOT just return the first option. Classify carefully.*
- User (prompted with a temperature of 1.0): *We're trying to test a new Alexa chatbot. I am going to give you two conversations, [labeled] conversation A and conversation B. In which conversation does the bot have a better conversation with the user, A or B? Justify your choice.*

  For guidance, a good conversation is where the bot shows their interest in the current topic, proves that it correctly comprehended what the user said, speaks precisely, and where both parties of the conversation speak for similar lengths. When making your decision, think about whether the conversation sounds like one that humans would have. Think about all of the conversations you see in books, movies, [and]real life, if that helps you get an idea of what a real conversation sounds like.

  Make sure your justification follows this guidance. It is of utmost importance that your reasoning stays consistent. The order in which I present the conversations must never affect your decision. There is a clear feature of one of these conversations that makes it a much better one than the other. You must find it, so your reasoning stays consistent.

  Remember, if I show you the same two conversations multiple times, your decision on which conversation was best must stay the same. Be consistent. Find the aspect of a conversation that makes it better than the other and stick with it.

  Conversation A: [Conversation A goes here]

  Conversation B: [Conversation B goes here]
  Classification:(A/B):
- Then, the Assistant replies with its selected conversation and a justification.
- User (prompts again with temperature 0): *We're trying to test a new Alexa chatbot. I am going to give you two conversations, [labeled] conversation A and conversation B.*

  Based on your previous analysis of these conversations, in which conversation does the bot have a better conversation with the user, A or B?

  Remember, you just analyzed these conversations a moment ago. Use that analysis to make this final classification.

Now, just give your classification result. You only need to say A or B, nothing more.

Conversation A: [Conversation A goes here]

Conversation B: [Conversation B goes here]

Classification:(A/B):