Toward Interactive Dictation

Belinda Li[♠*] **Jason Eisner**[◇] **Adam Pauls**[◇] **Sam Thomson**[◇]
[♠]MIT CSAIL [◇]Microsoft Semantic Machines
[♠]bzl@mit.edu
[◇]{jason.eisner,adam.pauls,samuel.thomson}@microsoft.com

Abstract

Voice dictation is an increasingly important text input modality. Existing systems that allow both dictation and editing-by-voice restrict their command language to flat templates invoked by trigger words. In this work, we study the feasibility of allowing users to interrupt their dictation with spoken editing commands in *open-ended* natural language. We introduce a new task and dataset, TERTiUS, to experiment with such systems. To support this flexibility in real-time, a system must incrementally segment and classify spans of speech as either dictation or command, and interpret the spans that are commands. We experiment with using large pre-trained language models to predict the edited text, or alternatively, to predict a small text-editing program. Experiments show a natural trade-off between model accuracy and latency: a smaller model achieves 28% single-command interpretation accuracy with 1.3 seconds of latency, while a larger model achieves 55% with 7 seconds of latency.

1 Introduction

Speech can be preferable for text entry, especially on mobile devices or while the user's hands are occupied, and for some users for whom typing is always slow or impossible. While fast and accurate automatic speech recognition (ASR) is now ubiquitous (Kumar et al., 2012; Xiong et al., 2016; Chiu et al., 2018; Radford et al., 2022), ASR itself only *transcribes* speech. In practice, users may also wish to *edit* transcribed text. The ASR output might be incorrect; the user might have misspoken; or they might change their mind about what to say or how to phrase it, perhaps after seeing or hearing their previous version. Azenkot and Lee (2013) found that users with visual impairment spent 80% of time editing text vs. 20% dictating it.

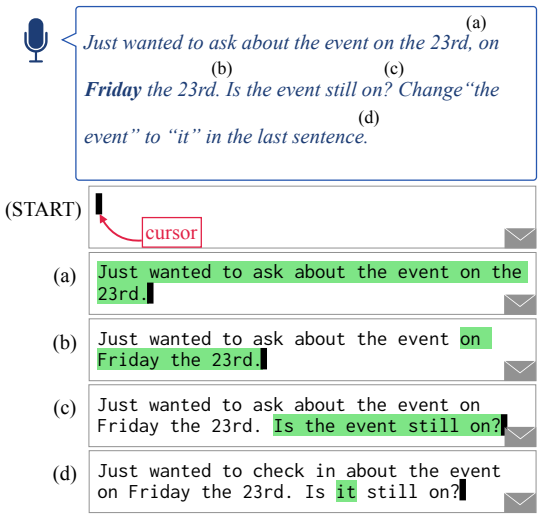

Figure 1: A user writes an email using speech input, interleaving dictation (a,c) and commanding (b,d). Top shows the continuous user utterance, while bottom shows the document state at each point of the utterance. Dictations are transcribed verbatim, while commands are interpreted and executed. Our system supports *open-ended commanding* (*i.e.*, b,d both invoke a `replace` operation but use vastly different phrasing).

In this work, we study the task of **interactive dictation**, in which users can both perform verbatim dictation and utter open-ended commands in order to edit the existing text, in a single uninterrupted speech stream. See Figure 1 for an example. Unlike commercial systems like Dragon (DNS; Nuance, 1997, 2022) and dictation for Word (Microsoft, 2022) that require reserved trigger words for commanding, the commands in our data are invoked using unrestricted natural language (NL). For example, in Figure 1, both (b) and (d) invoke `replace` commands, but (d) uses nested syntax to specify both an edit action and location, while (b) is implicit (as natural speech repairs often are).

In interactive dictation, users do not need to memorize a list of specific trigger words or templates in order to invoke their desired functionality. A dictation system should be as intuitive as dic-

[*] Work performed during a research internship at Microsoft Semantic Machines.

tating to a *human* assistant—a situation in which people quite naturally and successfully intersperse speech repairs and commands with their dictation. Beyond eliminating the learning curve, letting users speak naturally should also allow them to focus on what they want to say, without being repeatedly distracted by the frustrating separate task of getting those words into the computer.

Because we accept unrestricted NL for commands, both *segmentation* and *interpretation* become nontrivial for a system to perform.[1] Segmentation requires capturing (sometimes subtle) changes in intent, and is especially difficult in cases where command boundaries do not align with ASR boundaries.[2] We collect a dataset of 1320 documents dictated in an interactive environment with live, incremental ASR transcription and Wizard-of-Oz–style interpretation of user commands. Annotators were not told a set of editing features they were allowed to use, but simply instructed to make their commands understandable and executable by a hypothetical human helper. Collection required designing a novel data collection interface. Both the interface and dataset will be publicly released to help unlock further work in this area.[3]

Finally, we experiment with two strategies for implementing the proposed system: one that uses a pre-trained language model to directly predict the edited text given unedited text and a command, and another that interprets the command as a program specifying how to edit. Predicting intermediate programs reduces latency because the programs are short, at the expense of accuracy. This strategy also requires additional work to design and implement a set of editing functions and annotate commands with programs that use these functions.

For each strategy, we also experimented with two choices of pre-trained language model: a small fine-tuned T5 model and a large prompted GPT3 model. Using the smaller model significantly improves latency, though again at the cost of accuracy.

In summary, our contributions are: (1) a novel *task* (interactive dictation), (2) a novel *data collection interface* for the task, with which we collect a new *dataset*, and (3) a *system* that implements said task, with experiments and analysis.

2 Background & Related Work

Many modern speech input tools only support direct speech-to-text (*e.g.*, Radford et al., 2022). Occasionally, these models also perform disfluency correction, which includes removing filler words (e.g., *um*), repeated words, false starts, etc. (*e.g.*, Microsoft Azure, 2022). One form of disfluency that has received particular attention is speech repair, where the speaker corrects themself mid-utterance. For example, *let's chat tomorrow uh I mean Friday* contains a speech repair, where the user corrects "tomorrow" with "Friday." The repaired version of this should be *let's chat Friday*. Prior work has collected datasets and built systems specifically for speech repair (Heeman and Allen, 1994, 1999; Johnson and Charniak, 2004). Additionally, ASR systems themselves make errors that humans may like to correct post-hoc; there has been work on correcting ASR errors through *respeaking* misdetected transcriptions (McNair and Waibel, 1994; Ghosh et al., 2020; Vertanen and Kristensson, 2009; Sperber et al., 2013).

Beyond disfluencies that were not automatically repaired but were transcribed literally, humans must fix many other mistakes while dictating. They often change their mind about what to say—the human writing process is rarely linear—and ASR itself commonly introduces transcription errors. Most systems require the user to manually fix these errors through keyboard-and-mouse or touchscreen editing (*e.g.*, Kumar et al., 2012), which can be inconvenient for someone who already relies on voice for dictation. Furthermore, most commercial systems that support editing through speech (DNS, Word) require templated commands. Thus, while speech input is often used to write short-form, imprecise text (*e.g.*, search queries or text messages), it is not as popular as it might be, and it is used less when writing longer and more precise documents.

In our work, we study making edits through spoken natural language commands. Interpreting flexible natural language commands is a well-studied problem within NLP, with work in semantic parsing (Zelle and Mooney, 1993; Zettlemoyer and Collins, 2009; Artzi and Zettlemoyer, 2013), instruction-following (Chen and Mooney, 2011;

[1] In template-based systems, by contrast, commands can be detected and parsed using regular expressions. An utterance is considered a command if and only if it matches one of these regular expressions.

[2] In Figure 1, for example, we must segment the first sentence into two parts, a dictation (*"Just wanted to ask about the event on the 23rd"*) and a command (*"on Friday the 23rd"*). ASR can also *overpredict* boundaries when speakers pause in the middle of a sentence. For example, in our data *"Change elude mansion to elude mentioned."* was misrecognized by MSS as *"Change. Elude mansion to elude mentioned."*

[3] https://aka.ms/tertius

Branavan et al., 2009; Tellex et al., 2011; Anderson et al., 2018; Misra et al., 2017), and task-oriented dialogue (Budzianowski et al., 2018). Virtual assistants like Siri (Apple, 2011), Alexa (Amazon, 2014), and Google Assistant (Google, 2016) have been built to support a wide range of functionalities, including interacting with smart devices, querying search engines, scheduling events, etc. Due to advances in language technologies, modern-day assistants can support flexible linguistic expressions for invoking commands, accept feedback and perform reinterpretation (Semantic Machines et al., 2020), and work in an online and incremental manner (Zhou et al., 2022). Our work falls in this realm but: (1) in a novel interactive dictation setting, (2) with unrestricted commanding, and (3) where predicting boundaries between dictations and commands is part of the task.

Recently, a line of work has emerged examining how large language models (LLMs) can serve as collaborative writing/coding assistants. Because of their remarkable ability to generate coherent texts over a wide range of domains and topics, LLMs have proven surprisingly effective for editing, elaboration, infilling, etc., across a wide range of domains (Malmi et al., 2022; Bavarian et al., 2022; Donahue et al., 2020). Though our system also makes use of LLMs, it supports a different mode of editing than these prior works. Some works use edit models for other types of sequence-to-sequence tasks (e.g. summarization, text simplification, style transfer) (Malmi et al., 2019; Dong et al., 2019; Reid and Zhong, 2021), while others use much coarser-grained editing commands than we do, expecting the LLM to (sometimes) generate new text (Bavarian et al., 2022; Zhang et al., 2023). In addition to these differences, our editing commands may be misrecognized because they are spoken, and may be misdetected/missegmented because they are provided through the same channel as text entry.

3 Task Framework

We now formalize our interactive dictation setting. A user who is editing a document speaks to a system that both transcribes user dictation and responds to user commands. This process results in a **interactive dictation trajectory**—a sequence of timestamped events: the user keeps speaking, several trained modules keep making predictions, and the document keeps being updated.

Supervision could be provided to the predictive modules in various ways, ranging from direct supervision to delayed indirect reward signals. In this paper, we collect supervision that can be used to bootstrap an initial system. We collect **gold trajectories** in which every prediction is correct—except for ASR predictions, where we preserve the errors since part of our motivation is to allow the user to fix dictation errors.[4] All predictions along the trajectory are provided in the dataset.

Our dataset is not completely generic, since it assumes that certain predictive modules will exist and interact in particular ways, although it is agnostic to how they make their predictions. It is specifically intended to train a system that is a pipeline of the following modules (Figure 2):

(a) ASR As the user speaks, the ASR module proposes transcripts for spans of the audio stream. Due to ASR system latency, each ASR result normally arrives some time *after* the end of the span it describes. The ASR results are transcripts of successive disjoint spans of the audio, and we refer to their concatenation as the **current transcript** ($\mathcal{U}$ in Figure 2(a)).

(b) Segmentation When the current transcript changes, the system can update its segmentation. It does so by partitioning the current transcript $\mathcal{U}$ into a sequence of segments u_i, labeling each as being either a **dictation** or a **command**.

(c) Normalization *(optional)* Each segment u_i can be passed through a normalization module, which transforms it from a literal transcript into clean text that should be inserted or interpreted. This involves speech repair as well as text normalization to handle orthographic conventions such as acronyms, punctuation, and numerals.

While the module (a) may already attempt some version of these transformations, an off-the-shelf ASR module does not have access to the document state or history. It may do an incomplete job and there may be no way to tune it on gold normalized results. This normalization module can be trained to finish the job. Including it also ensures that our gold trajectories include the intended normalized text of the commands.

(d) Interpretation Given a document state d_{i-1} and a segment u_i, the interpretation module predicts the new document state d_i that u_i is meant

[4] In module (c) below, we predicted repairs for command segments, so the gold trajectory interprets accurate clean text for commands. But we did not predict repairs for dictation segments, so their errors persist even in the gold trajectories.

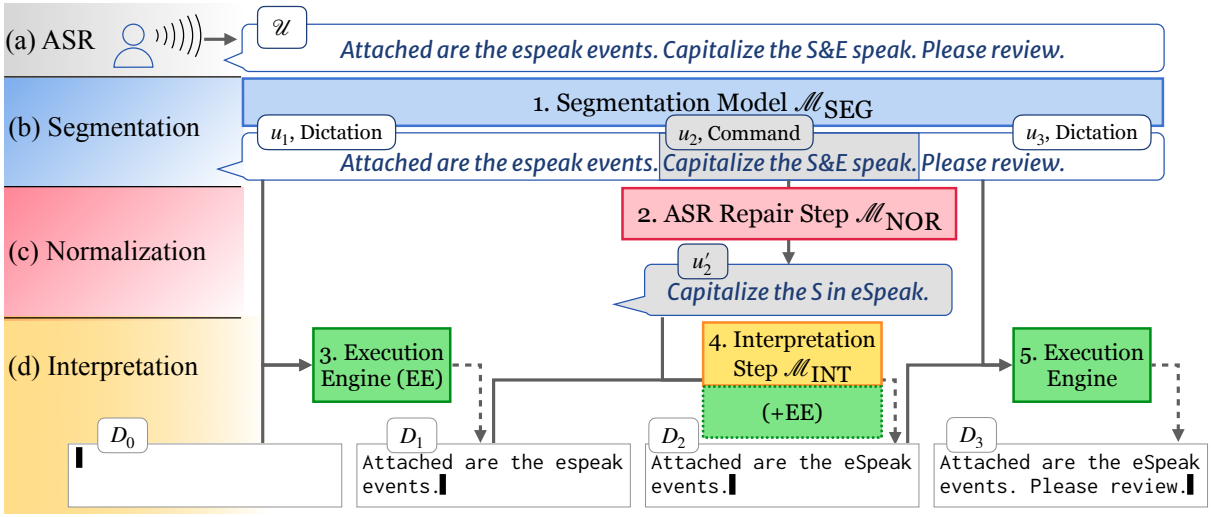

Figure 2: Diagram of an interactive dictation system. First, the ASR system (a) transcribes speech, which the segmentation system (b) parses into separate dictation and command segments. Next, an optional normalization module (c) fixes the any ASR or speech errors in the segment. Finally, the interpretation system (d) returns the result of each operation. On the right is the concrete instantiation of *our* system.

to achieve.[5] The document is then immediately updated to state d_i; the change could be temporarily highlighted for the user to inspect. Here d_{i-1} is the result of having already applied the updates predicted for segments $u_1, \ldots, u_{i-1}$, where d_0 is the initial document state. Concretely, we take a **document state** to consist of the document content together with the current cursor position.[6]

When u_i is a dictation segment, no prediction is needed: the state update simply inserts the current segment at the cursor. However, when u_i is a command segment, predicting the state update that the user wanted requires a text understanding model. Note that commands can come in many forms. Commonly they are imperative commands, as in Figure 1d. But one can even treat speech repairs such as Figure 1b as commands, in a system that does not handle repairs at stage (a) or (c).

Rather than predict d_i directly, an alternative design is to predict a program p_i and apply it to d_{i-1} to obtain d_i. In this case, the gold trajectory in our dataset includes a correct program p_i, which represents the intensional semantics of the command u_i (and could be applied to different document states).

[5]This prediction can also condition on earlier segments, which provide some context for interpreting u_i. It might also depend on document states other than d_{i-1}—such as the state or states that were visible to the user while the user was actually uttering u_i, for example.

[6]The cursor may have different start and end positions if a span of text is selected, but otherwise has width 0. For example, the document state d_1 in Figure 2 is $(\textit{"Attached are the espeak events."}, (31, 31))$.

Change Propagation The ASR engine we use for module (a) sometimes revises its results. It may replace the most recent of the ASR results, adding new words that the user has spoken and/or improving the transcription of earlier words. The engine marks an ASR result as **partial** or **final** according to whether it will be replaced.[7]

To make use of streaming partial and final ASR results, our pipeline supports change propagation. This requires the predictive modules to compute additional predictions. If a module is notified that its input has changed, it recomputes its output accordingly. For example, if module (a) changes the current transcript, then module (b) may change the segmentation. Then module (c) may recompute normalized versions of segments that have changed. Finally, module (d) may recompute the document state d_i for all i such that d_{i-1} or u_i has changed.

The visible document is always synced with the last document state. This sync can revert and replace the effects on the document of previous incorrectly handled dictations and commands, potentially even from much earlier segments. To avoid confusing the user with such changes, and to reduce computation, a module can freeze its older or more confident inputs so that they reject change notifications (Appendix B). Modules (b)–(d) could also adopt the strategy of module (a)—quickly return provisional results from a "first-pass" system with the freedom to revise them later. This could further

[7]Full details and examples can be found in Appendix A.1.

improve the responsiveness of the experience.

4 Dataset Creation

To our knowledge, no public dataset exists for the task of interactive dictation. As our task is distinct from prior work in a number of fundamental ways (§2), we create a new dataset, TERTiUS.[8]

Our data collection involves two stages. First, a human **demonstrator** speaks to the system and provides the gold segmentations, as well as demonstrating the normalizations and document state updates for the command segments. Later, for each command segment, an **annotator** fills in a gold program that would yield its gold state update.

For a command segments, we update the document during demonstration using the demonstrated state updates—that is, they do double duty as *gold* and *actual* state updates. Thus, we follow a gold trajectory, as if the demonstrator is using an oracle system that perfectly segments their speech into dictations (though these may have ASR errors) versus commands, and then perfectly interprets the commands. A future data collection effort could instead update the document using the imperfect system that we later built (§5), in which case the demonstrator would have to react to cascading errors.

4.1 Collecting Interactive Dictation

We build a novel data collection framework that allows us to collect speech streams and record gold and actual events.

We used an existing ASR system, Microsoft Speech Services (**MSS**; Microsoft Azure, 2022). We asked the demonstrator to play both the role of the *user* (issuing the speech stream), and also the roles of the *segmentation*, *normalization*, and *interpretation* parts of the system (Figures 2b–d). Thus, we collect actual ASR results, while asking the demonstrator to demonstrate gold predictions for segmentation, normalization, and interpretation.

The demonstration interface is shown in Figure 3. demonstrators were trained to use the interface, and told during training how their data would be used.[9] A demonstrator is given a task of dictating an email into our envisioned system (shown in the yellow textbox). We collected data in three scenarios:

1. **Replicate doc:** Exactly recreate an email from the Enron Email Dataset (Klimt and Yang, 2004).[10]

2. **Elaborate doc:** Expand a terse description of an email into an full email. The exact wording of the full email is up to the demonstrator.

3. **Replicate segment:** Exactly recreate the post-state d_i of a single command segment u_i (randomly sampled from the already-collected Replicate doc and Elaborate doc data), starting from its pre-state d_{i-1}. This does not have to be done with a single command.

A demonstrator must then reach the target state (either exactly for Replicate doc or Replicate segment, or to their satisfaction for Elaborate doc), following these three steps:

Step 1 (ASR, segmentation) The demonstrator starts speaking, which gets transcribed in real time by the built-in ASR system into ASR results. As they speak, they demonstrate what the segmentation system should do by holding down a key whenever they are speaking a command (as opposed to dictating). They can specify consecutive commands by quickly releasing and re-pressing the key.[11] This gives us a list of time intervals when the key was held down. By matching these to the ASR timestamps, we identify the gold command segments in the ASR transcript. The remaining segments of the transcript are labeled as dictation.[12]

Step 2 (normalization) All labeled segments are displayed in the right column of the UI. After the demonstrator has finished speaking, they fill in the normalized text for each command segment. (The segment shows original and normalized text in the `ASR` and `Gold ASR` fields.)

Step 3 (interpretation) Finally, for each command segment, the demonstrator manually carries out the gold state update.[13] They do this by clicking on a command segment u_i in the right column, which pulls up the associated document state d_i in the left column. Initially d_i is set to equal the pre-state d_{i-1}, and the demonstrator edits it with their

[8] **T**ranscribing and **E**diting in **R**eal-**Ti**me with **U**nrestricted **S**peech. Named for the human amanuensis Tertius of Iconium.

[9] We met with demonstrators ahead of time and provided them with written instructions, which are in Appendix A.2.

[10] Preprocessing details can be found in Appendix A.3.

[11] We do not allow two dictation segments to be adjacent—that would be equivalent to one concatenated segment.

[12] More details on how the ASR results are combined/segmented can be found in Appendix A.1.

[13] For dictation segments, the system automatically computes the gold state update by inserting the segment at the selection. This segment is an actual ASR result and may contain errors.

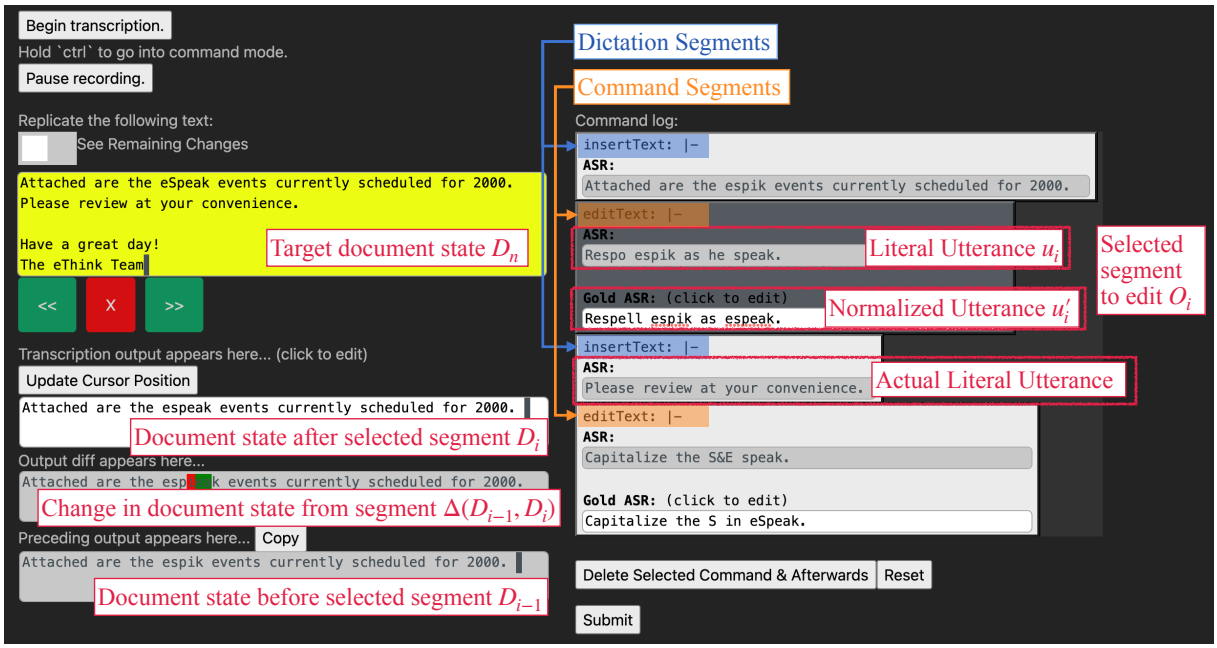

Figure 3: Data collection UI. Demonstrator speech is transcribed by a built-in ASR system. Demonstrators specify gold segmentations by pressing a key to initiate a command segment (`editText`) and releasing the key to initiate a dictation segment (`insertText`). The resulting transcribed segments appear in the `ASR` fields of the boxes in the right column. For a command segment, the demonstrator specifies the normalized version in the `Gold ASR` field, and demonstrates the command interpretation by editing the document post-state. Document states are shown in the left column: selecting a segment makes its post-state (and pre-state) appear there.

mouse and keyboard until it reflects the desired post-state after applying command u_i. For reference, the UI also displays the pre-state d_{i-1} and a continuously updated visual diff $\Delta(d_{i-1}, d_i)$).

Demonstrators can move freely among these steps, editing normalizations or state updates at any time, or appending new segments by speaking.[14]

We believe our framework is well-equipped to collect natural, flexible, and intuitive dictation and commanding data, for several reasons: (1) We do not restrict the capabilities of commands or the forms of their utterances, but instead ask demonstrators to command in ways they find most natural. (2) We simulate natural, uninterrupted switching between segments by making it easy for demonstrators to specify segment boundaries in real-time. (3) We collect a realistic distribution over speech errors and corrections by using an existing ASR system and asking demonstrators to replicate real emails. In the future, the distribution could be made more realistic if we sometimes updated the document by using predicted normalizations and state updates rather than gold ones, as in the DAgger imitation learning method (Ross et al., 2011).

[14]They are also allowed to back up and remove the final segments, typically in order to redo them.

4.2 Annotating Programs for Commands

After obtaining sequences of demonstrated dialogues using the above procedure, we extract each *command* segment and manually annotate it with a **program** p_i that represents the intensional semantics of the command. This program should in theory output the correct d_i when given d_{i-1} as input. Program annotation is done post-hoc with a different set of annotators from §4.1.

We design a domain-specific Lisp-like language for text-manipulating programs, and an execution engine for it. We implement a library consisting of composable *actions*, *constraints*, and *combinators*. A program consists of actions applied to one or more text targets, which are specified by contraints. Combinators allow us to create complex constraints by composing them. For example, in Figure 2, *Capitalize the S in eSpeak*, has the program

```
(capitalize
  (theText
    (and
      (like "S")
      (in (theText (like "eSpeak"))))))
```

where `capitalize` is the action, `(like "S")` and `(like "eSpeak")` are constraints, and `and` is a combinator. More examples are in Appendix A.4.

Task	Trajectories	Segments		
		Dict.	Cmd.	Total
Replicate doc	372	473	1453	1926
Elaborate doc	343	347	473	820
Replicate segment	605	139	1299	1438
Total	1320	959	3225	4184

Table 1: Dataset size statistics.

4.3 Handling of partial ASR results

The current transcript sometimes ends in a partial ASR result and then is revised to end in another partial ASR result or a final ASR result. All versions of this transcript—"partial" and "final"—will be passed to the segmenter, thanks to change propagation. During demonstration, we record the gold labeled segmentations for all versions, based on the timing of the demonstrator's keypresses.

However, only the segments of the "final" version are shown to the demonstrator for further annotation. A segment of a "partial" version can simply copy its gold normalized text from the segment of the "final" version that starts at the same time. These gold data will allow us to train the normalization model to predict a normalized command based on partial ASR results, when the user has not yet finished speaking the command or the ASR engine has not yet finished recognizing it.

In the same way, a command segment u_i of the "partial" version could also copy its gold document post-state d_i and its gold program p_i from the corresponding "final" segment. However, that would simply duplicate existing gold data for training the interpretation module, so we do not include gold versions of these predictions in our dataset.[15]

4.4 Dataset details & statistics

In the first stage (§4.1), eleven human demonstrators demonstrated 1372 interactive dictation trajectories (see Table 1 for details). In the second stage (§4.2), two human annotators annotated programs for 868 commands.[16] The dataset was then split into training, validation, and test sets with 991 training trajectories (consisting of 3199 demonstrated segments), 173 validation trajectories (562 segments), and 156 test trajectories (423 segments).

All demonstrators and annotators were native English speakers. The dataset is currently only English, and the editor supports unformatted plain text. However, the annotation framework could handle other languages that have spoken and written forms, and could be extended to allow formatted text.

A key goal of our system is flexibility. We quantify how well TERTiUS captures flexibility by measuring the *diversity* of natural language used to invoke each state change.[17] We count the number of distinct first tokens (mainly verbs) used to invoke each action. These results are reported in Table 4 in the Appendix, alongside a comparison with DNS.[18] We see that TERTiUS contains at least 22 ways to invoke a `correction`, while DNS supports only 1. In short, these results show that doing well on TERTiUS requires a much more flexible system that supports a wider array of functions and ways of invoking those functions than what existing systems provide.

5 Modeling & Training

The overall system we build for interactive dictation follows our pipeline from Figure 2 and §3:

1. A **segmentation model** $\mathcal{M}_{\texttt{SEG}}$ takes the current transcript $\mathcal{U}$, and predicts a segmentation $u_1, \ldots, u_n$, simultaneously predicting whether each u_i corresponds to a *dictation* or *command* segment.

2. Each dictation segment is directly spliced into the document at the current cursor position.

3. For each command segment:

   (a) A **normalization model** $\mathcal{M}_{\texttt{NOR}}$ predicts the normalized utterance u_i', repairing any ASR misdetections.

   (b) An **interpretation model**, $\mathcal{M}_{\texttt{INT(state)}}$ or $\mathcal{M}_{\texttt{INT(program)}}$, either: 1. directly predicts the end state of the command d_i, or 2. predicts the command *program* p_i, which is then executed to d_i by the

[15] The gold pre-state d_{i-1} may occasionally be different, owing to differences between the two versions in earlier dictation segments. In this case, the interpretation example would no longer be duplicative (because it has a different input). Unfortunately, in this case it is no longer necessarily correct to copy the post-state d_i, since some differences between the two versions in the pre-state might need to be preserved in the post-state.

[16] The rest of the programs were auto-generated by GPT3. See details in Appendix C.2.

[17] The system we build can theoretically support more flexibility than what is captured in TERTiUS. However, for TERTiUS to be a useful testbed (and training set) for flexibility, we would like it to be itself diverse.

[18] We also measure the diversity of state changes captured by TERTiUS in Appendix A.5.

execution engine. We experiment with both types of interpretation model.

Below we describe the specific models we use.

5.1 Segmentation

The segmentation model partitions $\mathcal{U}$ into segments u_i, each of which is labeled by m_i as being either dictation or command:

$$\begin{aligned} \mathcal{M}_{\texttt{SEG}}(\mathcal{U}) &= [(u_0, m_0), \cdots, (u_n, m_n)], \\ \text{s.t. } \mathcal{U} &= u_0 + u_1 + \cdots + u_n \\ m_i &\in \{\text{command}, \text{dictation}\} \end{aligned} \tag{1}$$

Concretely, the segmentation model does this using BIOES tagging (Jurafsky and Martin, 2009, Chapter 5). Here each command is tagged with a sequence of the form `BI*E` ("beginning, inside, ..., inside, end") or with the length-1 sequence `S` ("singleton"). Maximal sequences of tokens tagged with `O` ("outside") then correspond to the dictation segments. Note that two dictation segments cannot be adjacent. We implement the segmentation model as a T5-base encoder (Raffel et al., 2022) followed by a two-layer MLP prediction module. More details on why each tag is necessary and how we trained this model can be found in Appendix C.1.

5.2 Normalization and Interpretation

For each u_i that is predicted as a command segment, we first predict the normalized utterance u_i',[19]

$$\mathcal{M}_{\texttt{NOR}}(d_{i-1}, u_i) = u_i'. \tag{2}$$

We then interpret u_i' in context to predict either the document state d_i or an update program p_i.

$$\begin{aligned} \mathcal{M}_{\texttt{INT(state)}}(d_{i-1}, u_i') &= d_i, \\ \mathcal{M}_{\texttt{INT(program)}}(d_{i-1}, u_i') &= p_i. \end{aligned} \tag{3}$$

We then update the document state accordingly.

We experiment with two ways of implementing the two steps: we either fine-tune two separate T5-base models (Raffel et al., 2022) that run in a pipeline for each command, or we prompt GPT3 (Brown et al., 2020)[20] to generate both the normalized utterance[21] and the interpretation output in a single inference step. Training and prompting details can be found in Appendix C.2.

[19] Note that the normalization step additionally conditions on the state d_{i-1}, allowing it to consider what command would have been sensible in this context. Concrete examples are given in Appendix D.2.

[20] Specifically, the `text-davinci-003` model.

[21] Although the normalized utterance is not used for the final state prediction, early experiments indicated that this auxiliary task helped the model with state prediction, possibly due to a chain-of-thought effect (Wei et al., 2022).

[22] See Appendix D for details.

[23] We disregard the cursor position in this evaluation.

6 Results

We evaluate the segmentation model in isolation, and the normalization and interpretation steps together. (Appendices D.2 and D.3 evaluate the normalization and interpretation steps in isolation.)

For simplicity, we evaluate the models only on current transcripts $\mathcal{U}$ that end in **final** ASR results (though at training time and in actual usage, they also process transcripts that end in **partial** ones).[22]

6.1 Segmentation

Metrics **Exact match (EM)** returns 0 or 1 according to whether the entire labeled segmentation of the final transcript $\mathcal{U}$ is correct. We also evaluate macro-averaged **labeled F1**, which considers how many of the gold labeled segments appear in the model's output segmentation and vice versa. Two labeled segments are considered to be the same if they have the same start and end points in $\mathcal{U}$ and the same label (dictation or command).

Results Segmentation results on an evaluation dataset of transcripts $\mathcal{U}$ (see Appendix D.1) are shown in the top section of Table 2. All results are from single runs of the model. The model performs decently on TERTiUS, and in some cases is even able to fix erroneous sentence boundaries detected by the base ASR system (Appendix D.1.2). However, these cases are also difficult for the model: a qualitative analysis of errors find that, generally, errors arise either when the model is misled by erroneous over- and under-segmentation by the base ASR system, or when commands are phrased in ways similar to dictation. Examples are in in Appendix D.1.1.

6.2 Normalization & Interpretation

Metrics We evaluate normalization and interpretation in conjunction. Given a gold normalized command utterance u_i and the document's gold pre-state d_{i-1}, we measure how well we can reconstruct its post-state d_i. We measure **state exact match (EM)**[23] between the predicted and gold post-states. If the interpretation model predicts

	Metric	T5		GPT3	
Segmentation	F1	90.9%		-	
	Segmentation EM	85.3%		-	
	Runtime (s/it)	0.097		-	
		prog	**state**	**prog**	**state**
ASR Repair + Interpretation	State EM	28.3%	29.5%	38.6%	55.1%
	Program EM	28.3%	-	41.9%	-
	Runtime (s/it)	1.28	3.46	5.32	6.92

Table 2: We evaluate segmentation (top) and the ASR repair and interpretation components jointly (bottom). We report accuracy metrics (F1, EM) as well as runtime (in seconds per example). Segmentation is relatively fast and performs decently. For ASR repair and interpretation, we experiment with a fine-tuned T5 vs. a prompted GPT3 model, each outputting either the end state (state) or a program to carry out the command (prog).

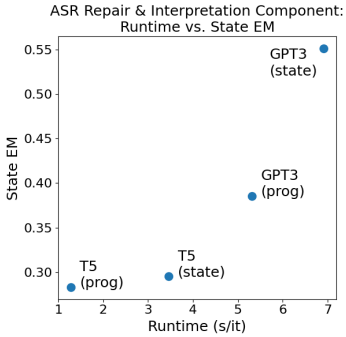

Figure 4: Runtime vs. State EM. GPT3 models produce more accurate state updates than T5, but use an unreasonable amount of time. Directly predicting the updated documents is more often correct than predicting update programs, again at the cost of time.

intermediate programs, then we also measure **program exact match (EM)** between the predicted program and the gold program.

Results The bottom of Table 2 shows these results. All results are from single runs of the model. GPT3 generally outperforms T5, likely due to its larger-scale pretraining. When we evaluated ASR repair and interpretation separately in Appendices D.2 and D.3, we found that GPT3 was better than T5 at both ASR repair and interpretation.

Furthermore, we find that *both GPT3 and T5 are better at directly generating states* (55.1 vs. 38.6 state EM and 29.5 vs. 28.3 state EM). However, the gap is larger for GPT3. We suspect that GPT3 has a better prior over well-formed English text and can more easily generate edited documents d directly, without needing the abstraction of an intermediate program. T5-base, on the other hand, finds it easier to learn the distinctive (and more direct) relationship between u and the short program p.

Other than downstream data distribution shift, we hypothesize that program accuracy is lower than state accuracy because the interpretation model is trained mostly on *auto-generated* program annotations, and because the execution engine is imperfect. We anticipate that program accuracy would improve with more gold program annotations and a better execution engine.

6.3 Efficiency

Table 2 reports runtimes for each component. This allows us to identify bottlenecks in the system and consider trade-offs between model performance and efficiency. We see that segmentation is generally quick and the ASR repair and interpretation steps are the main bottlenecks. The T5 model also runs much faster than the GPT3 model,[24] despite performing significantly worse, indicating a trade-off between speed and accuracy.

Figure 4 shows that by generating programs instead of states, we achieve faster runtimes (as the programs are shorter), at the expense of accuracy.

7 Conclusion

Most current speech input systems do not support voice editing. Those that do usually only support a narrow set of commands specified through a fixed vocabulary. We introduce a new task for *flexible invocation* of commands through natural language, which *may be interleaved with dictation*. Solving this task requires both *segmenting* and *interpreting* commands. We introduce a novel data collection framework that allows us to collect a pilot dataset, TERTiUS, for this task. We explore trade-offs between model accuracy and efficiency. Future work can examine techniques to push out the Pareto frontier, such as model distillation to improve speed and training on larger datasets to improve accuracy. Future work can also look at domains outside of (work) emails, integrate other types of text transformation commands (*e.g.*, formatting), and may allow the system to respond to the user in ways beyond updating the document.

[24] Note that GPT3 is called via an external API, while T5 is run on a local GPU. GPT3 runtimes thus include an unknown communication overhead, which will not be present when run on local hardware.

8 Limitations

TERTiUS is a pilot dataset. In particular, its test set can support segment-level metrics, but is not large enough to support reliable dialogue-level evaluation metrics. Due to resource constraints, we also do not report inter-annotator agreement measurements. While we made effort to make our interface low-friction, the demonstration setting still differs from the test-time scenario it is meant to emulate, and such a mismatch may also result in undesired data biases. Because our dialogues were collected before having a trained interpretation model, trajectories always follow gold interpretations. Because of this, the main sources of errors are ASR misdetections or user speech errors. In particular, TERTiUS contains data on: 1. misdetections and speech errors in transcription, and how to fix them through commands, 2. misdetections and speech errors in edits, and what intent they correspond to. We leave to future work the task of addressing semantic errors and ambiguities which result from incorrect interpretation of user intent. Some of these limitations can be addressed by incorporating trained models into the demonstration interface, which will allow faster demonstration, and capture trajectories that include actual system (non-gold) interpretations.

Though the trained system runs, we have not done user studies with it because it is not production-ready. The T5-base models are efficient enough, but the prompted GPT3 model is too slow for a responsive interactive experience. Neither model is accurate enough at interpretation. We welcome more research on this task!

When a human dictates to another human, interleaved corrections and commands are often marked prosodically (by pitch melody, intensity, and timing). Our current system examines only the textual ASR output; we have given no account of how to incorporate prosody, a problem that we leave to future work. We also haven't considered how to make use of speech lattices or n-best lists, but they could be very useful if the user is correcting our mistranscription—both to figure out what text the user is referring to, and to fix it.

9 Impact Statement

This work makes progress toward increasing accessibility for those who cannot use typing inputs. The nature of the data makes it highly unlikely that artifacts produced by this work could be used (intentionally or unintentionally) to quickly generate factually incorrect, hateful, or otherwise malignant text.

The fact that all speakers in our dataset were native speakers of American English could contribute to exacerbating the already present disparity in usability for English vs. non-English speakers. Future work should look to expand the diversity of languages, dialects, and accents covered.

References

Amazon. 2014. Amazon Alexa.

Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid, Stephen Gould, and Anton Van Den Hengel. 2018. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3674–3683.

Apple. 2011. Siri.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Shiri Azenkot and Nicole B. Lee. 2013. Exploring the use of speech input by blind people on mobile devices. *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*.

Mohammad Bavarian, Angela Jiang, Heewoo Jun, and Henrique Pondé. 2022. New gpt-3 capabilities: Edit & insert. [Online; posted 15-March-2022].

S.R.K. Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. 2009. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 82–90, Suntec, Singapore. 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.

Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gašić. 2018. MultiWOZ - a large-scale multi-domain Wizard-of-Oz dataset for task-oriented dialogue modelling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5016–5026, Brussels, Belgium. Association for Computational Linguistics.

David L. Chen and Raymond J. Mooney. 2011. Learning to interpret natural language navigation instructions from observations. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, page 859–865. AAAI Press.

Chung-Cheng Chiu, Tara N. Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J. Weiss, Kanishka Rao, Ekaterina Gonina, Navdeep Jaitly, Bo Li, Jan Chorowski, and Michiel Bacchiani. 2018. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, page 4774–4778. IEEE Press.

Chris Donahue, Mina Lee, and Percy Liang. 2020. Enabling language models to fill in the blanks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2492–2501, Online. Association for Computational Linguistics.

Yue Dong, Zichao Li, Mehdi Rezagholizadeh, and Jackie Chi Kit Cheung. 2019. EditNTS: An neural programmer-interpreter model for sentence simplification through explicit editing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3393–3402, Florence, Italy. Association for Computational Linguistics.

Debjyoti Ghosh, Can Liu, Shengdong Zhao, and Kotaro Hara. 2020. Commanding and re-dictation: Developing eyes-free voice-based interaction for editing dictated text. *ACM Transactions on Computer-Human Interaction*, 27.

Google. 2016. Google Assistant.

Peter Heeman and James F Allen. 1994. Detecting and correcting speech repairs. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 295–302, Las Cruces. New Mexico State University.

Peter A. Heeman and James Allen. 1999. Speech repairs, intonational phrases and discourse markers: Modeling speakers' utterances in spoken dialog. *Computational Linguistics*, 25(4):527–572.

Mark Johnson and Eugene Charniak. 2004. A TAG-based noisy-channel model of speech repairs. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 33–39, Barcelona, Spain.

D. Jurafsky and J.H. Martin. 2009. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall series in artificial intelligence. Pearson Prentice Hall.

Bryan Klimt and Yiming Yang. 2004. The Enron corpus: A new dataset for email classification research. In *Proceedings of the 15th European Conference on Machine Learning*, ECML'04, page 217–226, Berlin, Heidelberg. Springer-Verlag.

Anuj Kumar, Tim Paek, and Bongshin Lee. 2012. Voice typing: A new speech interaction model for dictation on touchscreen devices. In *Proceedings of CHI, 2012,*, pages 2277–2286. ACM.

Eric Malmi, Yue Dong, Jonathan Mallinson, Aleksandr Chuklin, Jakub Adamek, Daniil Mirylenka, Felix Stahlberg, Sebastian Krause, Shankar Kumar, and Aliaksei Severyn. 2022. Text generation with text-editing models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Tutorial Abstracts*, pages 1–7, Seattle, United States. Association for Computational Linguistics.

Eric Malmi, Sebastian Krause, Sascha Rothe, Daniil Mirylenka, and Aliaksei Severyn. 2019. Encode, tag, realize: High-precision text editing. 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 5054–5065, Hong Kong, China. Association for Computational Linguistics.

Arthur E. McNair and Alex Waibel. 1994. Improving recognizer acceptance through robust, natural speech repair. In *Proc. 3rd International Conference on Spoken Language Processing (ICSLP 1994)*, pages 1299–1302.

Microsoft. 2022. Dictation for Microsoft Word.

Microsoft Azure. 2022. Cognitive speech services.

Dipendra Misra, John Langford, and Yoav Artzi. 2017. Mapping instructions and visual observations to actions with reinforcement learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1004–1015, Copenhagen, Denmark. Association for Computational Linguistics.

Nuance. 1997. Dragon NaturallySpeaking.

Nuance. 2022. Dragon Speech Recognition Solutions.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie

Bai, and Soumith Chintala. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., Red Hook, NY, USA.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2022. Robust speech recognition via large-scale weak supervision.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2022. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1).

Machel Reid and Victor Zhong. 2021. LEWIS: Levenshtein editing for unsupervised text style transfer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3932–3944, Online. Association for Computational Linguistics.

Stephane Ross, Geoff J. Gordon, and J. Andrew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of AISTATS*.

Semantic Machines, Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Matthias Sperber, Graham Neubig, Christian Fügen, Satoshi Nakamura, and Alex Waibel. 2013. Efficient speech transcription through respeaking. pages 1087–1091.

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R. Walter, Ashis Gopal Banerjee, Seth Teller, and Nicholas Roy. 2011. Understanding natural language commands for robotic navigation and mobile manipulation. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, page 1507–1514. AAAI Press.

Keith Vertanen and Per Ola Kristensson. 2009. Automatic selection of recognition errors by respeaking the intended text. In *ASRU '09: IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 130–135.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig. 2016. Achieving human parity in conversational speech recognition.

John M. Zelle and Raymond J. Mooney. 1993. Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI'93, page 817–822. AAAI Press.

Luke Zettlemoyer and Michael Collins. 2009. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 976–984, Suntec, Singapore. Association for Computational Linguistics.

Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Jiawei Zhou, Jason Eisner, Michael Newman, Emmanouil Antonios Platanios, and Sam Thomson. 2022. Online semantic parsing for latency reduction in task-oriented dialogue. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1554–1576, Dublin, Ireland. Association for Computational Linguistics.

A Dataset

A.1 ASR results

Types of segments Below we describe the types of ASR results we collect in TERTiUS. As dialogues are uttered, we obtain a stream of timestamped partial and full ASR results from MSS. Examples of partial and full ASR results can be found below:

0:00.00: *attached*
0:00.30: *attached is*
0:00.60: *attached is the*
0:01.05: *attached is the draft*
0:02.15: *Attached is the draft.*

The first four lines are partial ASR results u^{partial} that are computed quickly and returned by MSS in real time as the user is speaking. The last line is the final ASR result, which takes slightly longer to compute, but represents a more reliable and polished ASR result. After a final result u^{final} has been computed, it obsolesces prior partial ASR results.

While not used in present experiments, collecting partial ASR results enables building an incremental system that can be faster and more responsive in real time; rather than waiting for ends of sentences to execute commands, a system can rely on partial ASRs to anticipate commands ahead of time (akin to Zhou et al. (2022)). Collecting timing information is also helpful for evaluating the speed of our system: the system runtime continges on the rate at which it obtains new ASR results and how long it takes to process them.

Furthermore, MSS additionally returns n-best lists for each final ASR results. These are a list of candidate final ASRs that may feasibly correspond with user audio, *e.g.*,

Attached is the draft.
Attached his draft.
Attacked is the draft.
...

Aggregation segments For long user audio streams, partial and final results are returned sequentially, each describing roughly a single sentence. The most recent ASR result is concatenated together with the previous history of final ASR results, to return the full partial or final ASR result for the entire stream. For example, after the user utters the first sentence in the example above, the user may continue by saying:

please
please re
please review
please review win
please review when pause
please review when possible
Please review when possible.

We concatenate each of these new ASR results with the previous final ASR results to obtain the current transcript $\mathcal{U}$ (see §3),which evolves over time as follows:

Attached is the draft. please
Attached is the draft. please re
Attached is the draft. please review
Attached is the draft. please review win
Attached is the draft. please review when pause
Attached is the draft. please review when possible
Attached is the draft. Please review when possible.

Segmenting ASR results into Segments During Annotation During annotation (§4.1), all these partial and final ASR results get mapped to segments, forming u_i^{final} and u_i^{partial}. This is done by identifying the *timestamp of each token* within each partial and final result. For example, in the example ASR results sequence at the beginning of this section A.1, suppose the user specifies an segment boundary at time 0:00.45, (separating "*Attached is*" from "*the draft.*"). We get the following ASR results for the first segment:

attached
attached is
Attached is

(we refer to the first two as partial ASRs for the segment, as they are derived from partial ASR, and the third as the final ASR for the segment), and the following ASR results for the second segment:

the
the draft
the draft.

A.2 Annotation Instructions (§4.1)

The full text of written instructions given to annotators during the first round of annotation (§4.1) is provided below:

1. **Transcribing**

   Your goal is to replicate the prompt in the target box verbatim / expand the prompt in the yellow textbox into a coherent email, starting from the given (potentially non-empty) starting document in the 'Transcription output' box. You are expected to do so using a series of speech-to-text transcriptions and commands. Try to use the starting document as much as possible (i.e. do not delete the entire document and start over).

   You can easily see what changes are to be made by toggling the 'See Diff View' button. Once that mode is on, the text you need to add will be highlighted in green, while the text you need to delete will by highlighted in red. Once there is no colored text, your text box matches the target text box and you are done.

   Begin this process by hitting the 'Begin transcription' button. This will cause a new 'insertText' command to appear in the command log on the right.

   You are now in transcription mode. Whatever you say will appear in the 'Transcription output' box.

2. **Editing**

   You can fix mistakes in transcription, add formatting, etc. through adding 'editText' commands.

   Hold down 'ctrl' on your keyboard to issue a new 'editText' command.

   While holding down 'ctrl' you will be in edit mode. In this mode, you can manually use mouse-and-keyboard to change the output. However, you must describe the edit you are making before you make it.

   Begin by describing your edit using your voice. Whatever you say now will appear in the editText ASR box, but not in the 'Transcription output'.

   Because the ASR system is imperfect, the textual description may be faulty. Fix any mistakes in the detected speech in the 'Gold ASR' box.

   Finally, manually edit the 'Transcription output' box to correspond the effect of your edit command.

   Note: It is important that you vocalize your change before making any edits to either 'Gold ASR' or 'Transcription output', as the ASR system stops recording as soon as you click into either one of these boxes.

3. **Undoing, Reseting, Submitting, & Saving**
   You can click on previous commands in the command log to revisit them. Note that if you edit the output associated with a 'editText' prior in the history, you will erase the changes associated with subsequent 'editText' operations.

   If you would like to undo some portion of command log, you can use the 'Delete Selected Command & Afterwards' button. Simply click on the first command you would like to remove, then click the button to remove that command and all commands after it.

   You can clear the entire command log by hitting "Reset".

   If you would like to work on transcribing another target, use the green arrow keys below the target. This will present you with a new target while saving progress on your current target. To delete a target prompt, press the red 'X'.

   Once you are done editing, click "Submit" button.

   Please double-check each command before submission! In particular, commands will appear red if they are potentially problematic (e.g. they are not associated with any change to the underlying text). Please check to make sure there are no red commands that you do not intend to be there!

A.3 Target Text Preprocessing

For replicating **Enron** emails, we process emails from the Enron Email Dataset to create our target final states. We break the email threads into individual emails, filtering out email headers and non-well-formed emails (emails that are either less than 50 characters or more than 5000 characters long, or contain too many difficult-to-specify non-English symbols). Annotators also had the option to skip annotating certain emails, if they found the email too difficult to annotate.

A.4 Annotation Programs

Examples of programs can be found below:

Actions	Constraints & Combinators	
combineSentences	union	between
parenthesize	or	endsWith
allCaps	and	at
do	in	atStart
respell	nthToLast	atEnd
delete	nth	exactly
spell	findAll	hasSubstring
capitalize	thePosition	passage
combine	theText	line
quote	empty	sentence
lowercase	extra	parenthetical
move	nextTo	phrase
moveCursor	take	word
replace	contains	letter
insert	before	text
correction	after	like
	startsWith	alwaysTrue

Table 3: List of functions present in TERTiUS.

1. ASR: *Lower case the W in the word when.*
Program:

```
(lowercase
  (theText
    (and
      (like "W")
      (in
        (theText
          (and
            (word)
            (like "when")))))))
```

2. ASR: *Get rid of the space in between the two words off site and replace that with a -.*
Program:

```
(replace
  (theText
    (and
      (like " ")
      (between
        (theText (like "off"))
        (theText (like "site")))))
  "-")
```

A.5 Dataset Analysis

To assess the diversity of *state changes*, we quantify the number of distinct *actions*, *constraints*, and *constraint combinators* (see §4.2) that appear in the annotated programs. In Table 3, we list out all actions, constraints, and constraint combinators present in TERTiUS. TERTiUS contains at least 15 types of actions (and allows for action composition with sequential chaining operation `do`), with 34 types of constraint and constraint combinators.

In Table 4, we approximate the invocation diversity represented in TERTiUS, by measuring the number of distinct first tokens used to invoke each type of actions. For actions that overlap in function with ones supported by DNS, we also report a similar diversity metric against the full set of trigger words supported by DNS.[25]

Command action	# of distinct first tokens (TERTiUS)	# of distinct first tokens (DNS)
capitalize	12	2
replace	83	-
delete	22	5*
quote	2	1
parenthesize	3	1
do	44	-
insert	51	1
correction	22	1
lowercase	12	1
allCaps	8	1
spell	17	1
move	3	-
respell	1	-
combineSentences	7	-
moveCursor	3	1
combine	1	-

Table 4: Number of ways to invoke various commands, in terms of number of distinct first tokens used to invoke that command. Second column shows the number of distinct first invokation tokens as present in TERTiUS, while third column shows the number of distinct first invokation tokens for comparable commands supported by DNS.
*Counting *undo*, *backspace*, and *scratch that* as delete commands, despite being less general than our delete functionality (can only delete most recent tokens).

B Running Online

When running the system online in real time, we must consider efficiency and usability. We introduce a "commit point" that signifies that the system cannot re-segment, re-normalize, or re-interpret anything before that point. We only want to consider recent ASR results because the system quickly becomes inefficient as the dialogue length grows (the interpretation step, which is the bottleneck of the system, must run for every single command.) Furthermore, users often refer to and correct only recent dictations and commands; reverting early changes can have potentially large and undesirable downstream effects, leaving users potentially highly confused and frustrated.

Concretely, the commit point is implemented as the system treating the document state at that point as the new "initial state," so that it is unable to access segments and the history of document states from before that point. We implement this

[25] https://www.nuance.com/asset/en_us/collateral/dragon/command-cheat-sheet/ct-dragon-naturally-speaking-en-us.pdf

point so that it must coincide with the end of a final ASR result. We feed into the system this state as the initial state, and the entire sequence of ASR results starting from that point. All dictations and command segments returned by the model are executed in sequence from the commit point.

We decide to set a commit point based on system confidence and time since last commit. System confidence is derived from the confidences of each component model at each step of the prediction. We measure the system confidence of the *end state* predicted by the system, by summing the log-probabilities of: 1. the segmentation model result, (summing the log-probabilities of each BIOES tag predicted for each token), 2. the ASR repair model result for each command (log-probability of the resulting sentence), 3. the interpretation model result for each command (the log-probability of the end state or program). Once the system confidence exceeds a threshold τ_{commit}, we decide to commit immediately at that point. Otherwise, if we have obtained more than 4 final ASR results since the last commit, we must commit at our most confident point from within the last 4 turns.

C Model Training Details

In this section, we describe how we trained each component of the system. See §5 for a description of the inputs, outputs, and architecture of each model. Our final system is *incremental*, able to process both partial and final ASR results.

C.1 Segmentation Model

We use BIOES for the segmentation model. Note that we cannot just predict a binary command/dictation tag for each token, because it would be unable to discern two consecutive commands from one continuous command. Thus, we need to use **B** to specify the beginning of a new command segment. **E** is also necessary for the model to predict whether the final segment, in particular, is an incomplete and ongoing (requiring the ASR repair model to predict the future completion) or complete (requiring the ASR repair model to only correct errors).

We expect in the final online version of the end-to-end system, the segmentation model will: 1. run often, being able to accept and segment both partial and final ASR results, 2. run on only the most recent ASR, to avoid completely resegmenting an entire document that's been transcribed. Thus, we construct the training data for this model in a way to simulate these conditions. We extract all sequences of turns of length between 1 – 4 from TERTiUS (capping to at most 4 for condition 2), take their segments u, and concatenate them to simulate $\mathcal{U}$, asking the model to segment them back into their individual u. For the final turn of each chosen sequence, we include in the training data both the final ASR result and all partial ASR results. We fine-tune on this data with a learning rate of 1e-4 and batch size of 4 until convergence.

C.2 ASR Repair & Interpretation Models

Below we describe the concrete implementations and training details of each model:

T5 In the T5 implementation, both $\mathcal{M}_{\texttt{NOR}}$ and $\mathcal{M}_{\texttt{INT}}$ are T5-base encoder-decoder models.

As described in §4.4, we do not have annotations of programs for the full training split. Thus, we automatically generate the missing programs using GPT3.

We have an initial training reservoir that consists solely of data points with program annotations $\mathcal{D}_{\text{annot}}$. For each example in the remaining training set, we retrieve a subset of samples from $\mathcal{D}_{\text{annot}}$ to form the prompt. We also use GPT3 for this retrieval step[26].

We then annotate programs in the remaining training set in an iterative manner: as new programs are annotated, we use the execution engine to check whether it executes to the correct end state, and if so, we add it to $\mathcal{D}_{\text{annot}}$, such that future examples can include these programs in their prompt.

GPT3 In the GPT3 implementation, both the ASR repair and interpretation steps occur in a single inference step, with GPT3 being prompted to predict both outputs in sequence. Specifically, it is prompted with:

`[Input State:]`
d_{i-1}
`[Utterance ASR:]` u'_i
`[Gold Utterance:]` u_i
`[Final State:]`
d_i

The model is shown demonstrations in this format from the training data, then asked to infer, for each test sample, the highlighted portions from the non-highlighted portions.

[26] we compute similarity between two prompts by looking at the the similarity over next-token distributions when conditioned on each of the prompts

	Metric	T5	GPT3
ASR Repair	EM	47.3	70.7
Interpretation	Program EM	36.1	-
	State EM	33.7	54.2

Table 5: We evaluate the ASR repair and interpretation components in isolation. We experiment with a fine-tuned T5 vs. a prompted GPT3 model.

In the setting that we are predicting programs instead of end states, the final 2 lines are replaced with

`[Lispress:]` ℓ_i

D Results

D.1 Segmentation

We run all the error analyses in this section on a model trained and tested exclusively on the Replicate doc task (where annotators were asked to replicate emails from the Enron Email Dataset).

We do not evaluate the segmentation model on all of the transcripts that arise during a trajectory, many of which are prefixes of one another. Doing so would pay too little attention to the later segments of the trajectory. (F1 measure on the final transcript will weight all of the segments equally, but F1 measure on the earlier transcripts does not consider the later segments at all.)

Instead, we create an evaluation set of shorter transcripts. For each trajectory, we form its final full transcript by concatenating all of its final ASR result results. Each sequence of up to 4 consecutive gold segments of this full transcript is concatenated to form a short transcript that the segmentation model should split back into its gold segments. For example, if the full transcript consists of 8 gold segments, then it will have $8+7+6+5$ evaluation examples of 1 to 4 segments each.

D.1.1 Error Analysis

Below, we list some examples of segmentation errors ($[\cdot]$ is used to specify segment boundaries, yellow-highlighted segments correspond to command segments, while non-highlighted segments correspond to dictation segments).

1. **Input:** *Take out the word it. Before the word should. And then replace third with three.*
   True Segmentation: [*Take out the word it. Before the word should. And then replace third with three.*]
   Pred Segmentation: [*Take out the word it.*] [*Before the word should. And then replace third with three.*]

2. **Input:** *You learned. You lie not you learned.*
   True Segmentation: [*You learned.*] [*You lie not you learned.*]
   Pred Segmentation: [*You learned. You lie not you learned.*]

3. **Input:** *Skillings calendar is amazingly full! Let's shoot for one of the following.Skillings should be apostrophe s Let's schedule it ASAP.*
   True Segmentation: [*Skillings calendar is amazingly full! Let's shoot for one of the following.*] [*Skillings should be apostrophe s*] [*Let's schedule it ASAP.*]
   Pred Segmentation: [*Skillings calendar is amazingly full! Let's shoot for one of the following.Skillings should be apostrophe s Let's schedule it ASAP.*]

These examples illustrate two prototypical modes of errors: (i) the ASR system making erroneous judgments about sentence boundary locations, leading the segmentation model astray, and (ii) commands being phrased in ways that disguise them as dictations. The first example illustrate error type (i): the ASR system oversegments the input (which should've been a single sentence) into three separate sentences, consequently leading the segmentation system to believe "*Take out the word it*" and "*Before the word should...*" are separate commands. The second example illustrates error type (ii): "*You lie not you learned.*" is supposed to be a replace command indicating "*You learned*" should be replaced with "*You lie*", but it is not phrased as an explicitly command. Finally, the third example illustrates both error types: we see that the ASR system undersegments the input and combines the sentence "*Skillings should be apostrophe s*" with the sentence "*Let's schedule it ASAP*" without a period. Combined with the fact that "*Skillings should be apostrophe s*" is not issued explicitly as a command, this confuses the segmentation model into thinking that it is in fact part of the dictation.

D.1.2 Success Cases: Fixing Erroneous Segmentations

The above examples illustrated certain cases where the segmentation model was misled by erroneous ASR judgments about sentence boundary locations.

In some cases, however, the segmentation model is able to fix these judgements, as shown below:

1. **Input:** *Take out the extra space. In between the two words, but and should.*
   True/pred Segmentation: [*Take out the extra space. In between the two words, but and should.*]

2. **Input:** *Replace the period. With a comma after restructuring.*
   True/pred Segmentation: [*Replace the period. With a comma after restructuring.*]

D.2 ASR Repair

Metrics To measure the ASR repair step in isolation, we take noisy utterances u_i corresponding to each command and measure to what extent we are able to reconstruct the ground-truth utterance. We measure the percent of u_i for which our predicted repaired utterance exactly matches the ground-truth utterance (**EM**).

Results From Table 5, we see that the GPT3 model is much better at repairing speech disfluencies and ASR errors than the T5 model, achieving 70% EM. We suspect this is due to the fact that GPT3 was pretrained on much more (English) language data than T5, giving GPT3 a greater ability to produce grammatically coherent and permissible English sentences, and likely also a better sense of common disfluencies.

Qualitative Analysis Recall that we designed the ASR repair step to condition on not just the utterance u_i but the state d_{i-1}. This allows it take d_{i-1} into account when repairing u_i.

For example, when given the following utterance:

> *Delete the period after events.*

An ASR repair model that looks at ASR alone may not see any issue with this utterance. However, given the document state:

> Eric, I shall be glad to talk to you about it. The first three days of the next week would work for me. Vince.

(note the word *events* does not appear anywhere in this text), the ASR repair model realizes that the actual utterance should've been,

> *Delete the period after Vince.*

Indeed, the T5 ASR repair model is able to make the appropriate correction to this utterance.

D.3 Interpretation

Metrics To measure the interpretation step in isolation, we take normalized utterances u'_i corresponding to each command and measure to how well the interpretation model is able to reconstruct the ground-truth final state for the command d_i. We use the same set of metrics described in §6.2 (state EM, program EM). However, insteading of feeding the interpretation model ASR repair results, we feed in ground-truth utterances u.

Results We evaluate a T5 interpretation model that produces programs (which is then executed by our execution engine) and a GPT3 interpretation model that directly generates states. Results are reported in Table 5.

We can also compare these isolated interpretation results with the joint ASR and interpretation results reported in Table 2. Due to error propagation, the T5 model is ∼5–8% worse when asked to jointly perform ASR repair and interpretation from noisy ASR, than when simply asked to interpret normalized utterances. Surprisingly however, the GPT3 model performs nearly as well in the joint evaluation as the isolated evaluation. We suspect that even if the GPT3 ASR repair model does return the exactly normalized utterances, it is still able to reconstruct a semantically equivalent/similar command.

E Infrastructure and Reproducibility

We trained 220M-parameter T5-base model on a single NVIDIA Tesla A100 GPU machine. Each training run for each component of the model took at most a few hours (<8). We also prompted a 12B-parameter GPT3 model.

We used PyTorch (Paszke et al., 2019) and Huggingface Transformers (Wolf et al., 2020) for implementing and training T5-base models. We use OpenAI's API[27] for querying GPT3. We use the `text-davinci-003` model.

[27] https://beta.openai.com/