In Conversation, There Are No Errors

Actions on Google

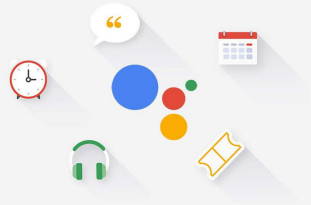

Table of contents

One of the hardest and most often neglected parts of designing conversational interfaces is knowing how to recover from so-called "no-match errors" (the user says something that you don't recognize) or "no-input errors" (the user says nothing at all).

Oftentimes, we commonly and mistakenly treat these error events as edge cases and handle them too simply, like apologizing and asking the same question again or handling them with an overly formulaic and prescriptive approach, which leads to a stilted experience at best and a really frustrating experience at worst.

People hear polite prompts such as "I didn't get that" or "I'm sorry I didn't understand", and the message they take away is "I don't understand anything" or "This technology doesn't work." This means recovering from such "errors" is critical to your users' experience and your app's success.

Errors as opportunities

There's no such thing as a query without intent. Users always want to do something, even if they don't overtly say so. To approach errors in a new way, treat them as new turns in the dialog with different conditions. Such cases can be opportunities to forge more meaningful exchanges with users by building trust and leveraging their innate expectations of how everyday conversations are supposed to work.

In human conversations, hesitations and corrections happen all the time. But in human-to-computer interactions, they cause timeouts and recognition errors. The difference is that people take cues from each other to get back on track, intuitively, and ***in real time.*** But with automated, manufactured conversations, corrections have to be planned out, designed, and programmatically accounted for ***ahead of time.***

The only way to do this while still maintaining a natural conversational flow is to treat such cases as inputs that don't, in and of themselves, lead to "errors." For starters, this means using prompts that give people credit for knowing how to talk.[1] From there, it means eliminating the bulk of your errors through preventative strategies, then developing a targeted strategy that fits each turn in the conversation and situational context.

Know what can go wrong

It takes a lot of things all working together to make a dialog successful: voice-signal processing, language parsing, audio-data transmission, software activation, and more. All the mechanics have to work properly just to capture the spoken input and return a relevant result. An unexpected input then produces an "error event," and that's when things get interesting.

Separating human conditions from the machine

The key thing to remember is that there's a difference between the technical conditions that trigger and respond to such events and what's actually happening at the same time from the user's perspective. From noise and interruptions, being cut off mid-sentence, hearing too many choices, or just being cooperative,[2] users are experiencing the "error" in a very real sense and not at all the same way the application logic is processing it.

From a technical standpoint, four basic things can go wrong:

1. Failure to get any input, either because there was none, or it wasn't detected. As a result, the system times out waiting for a response.
2. Input is received but not recognized or parsed, because of background noise or multiple people talking.
3. Input is recognized, but the app doesn't know how to handle it. For example, users might say, "I don't know, what can I do?", and your app parses the text correctly, but can't address the question appropriately.
4. Input is recognized but as the wrong thing—this can be the worst kind of error, because a user can head down the wrong path and the conversation can derail further as a result.

To begin tackling a solution to these, you can actually start by simply breaking down the problem further, into just two paths:

1. You didn't get any input (no-input error).
2. You got input, but you just weren't prepared to handle it (no-match error)

[1] https://developers.google.com/actions/design/unlocking-the-power-of-spoken-language
[2] https://developers.google.com/actions/design/be-cooperative

Now you know what you're trying to solve for programmatically. However, that's where the simplicity stops and a more strategic approach needs to take over, which is described in the following sections.

Develop strategies for handling errors

Let's look at how we can handle these errors and with robust strategies. You can implement some of these in tools such as API.AI,[3] some in code or fulfillment logic, and some with a combination of all these.

Effective prompting

Here are several prompting strategies for resolving errors.

Rapid re-prompt (without context)

- *"What was that?"*
- *"Say that again?"*

Rapid re-prompt (with context)

- *"Sorry, what time?"*
- *"I missed that number."*

Rephrase the question

- *"First, what's your favorite color?" → "What's your favorite color?"*
- *"Sure, what movie would you like to see?" → "To get started, what movie do you want to see?"*

Reframing the question

- *"What time is this for?" → "Sorry, what time?"*
- *"For when?" → "What time would you like to book this for?"*

Answering an unasked question

- *"I have your name and email from your account, so now all I need is your phone number."*
- *"You can give me the day, the time, or both."*

Being proactive

- *"I could put you down for 6 p.m. for now, does that work?*
- *"Do you want to finish this later?"*

Help in the moment

One important repair strategy involves preparing for users who get confused, didn't hear a question, or are unsure what to say. In these cases, you can apply preventative strategies, such as using intuitive language and well-crafted prompts. However, be ready anyway for people asking to hear something again (e.g. "can you say that again?", or a repeat intent), or saying something like "Help" or "I don't know."

Know when to quit

Another basic strategy to prevent frustration is to make it easier for users to leave the conversation if they haven't completed a task or validated a response, because they might need to stop for a variety of reasons. After all, life gets in the way sometimes. Being prepared for the user's departure is not only key to preventing errors, it's the right thing to do. It also can be an opportunity to re-engage with them later by letting users know how to come back and pick up where they left off, but without getting in their way.

Example (app quits):

App	*I'm thinking of a number. What's your first guess?*
App	*I didn't hear a number.*
App	*If you're still there, what's your guess?*
App	*We can stop here. Let's play again soon.*

Example (user quits):

User	Let's stop playing.
App	*Ok. Your score was 3 out of 5. Talk to you later.*

Robust non-error paths

A powerful notion to remember is that users who aren't encountering errors should feel like they're progressing. That way, if they do encounter an error later, they won't feel derailed.

[3] https://api.ai

Sound more human throughout

One way to sound natural and to "disguise" errors is by inserting variability to make the conversation more engaging, not just in error prompts but throughout the dialog. Use randomization and variable content in questions and responses to help mix things up.

Here are some useful strategies for making your app sound more human:

- Use a list of prompts and scale to any number of them without changing the code.
- Select from those prompts randomly.
- Combine prompts to create a large number of permutations.
- Add dynamic values by formatting prompts with placeholder symbols that are replaced at runtime: "Welcome, %s."
- Remember previous prompts and avoid using them again when randomly picking the next prompts.
- Track the number of errors, then adjust your prompts so they're more relevant to the most common errors.

Work to earn users' trust

Be prepared for basic questions that users might ask to poke at the system just to figure out what it can do. Think of it this way: to establish trust with a neighbor, you might borrow a cup of sugar before you ask to borrow the lawnmower. People want to see if the UI they're interacting with knows what they expect it to know.

Be proactive and leverage success

Reminding users how far they've come or that they don't have far to go to wrap things up helps to get them back on track.

Depending on your app's persona and how assertive it is, you may also want to take control of a situation to keep the conversation moving forward.

Best practices

Remember to treat input "errors" as natural parts of the conversation

→ **Don't treat technical error "events" as users misbehaving**

→ **Handle different types of error events with the appropriate strategy**

→ **Prevent errors by providing help in the moment**

→ **Know when to give up**

→ **Make the success path more robust to "disguise" errors**