Agent Completion Playbook

Introduction

Agent Completion (AC) tasks involve crafting realistic multi-turn conversations where an assistant uses a set of **function-calling tools** to help a user achieve a goal. As the conversation designer (trainer), you will simulate both the user and the assistant and the assistant's tool calls (only when the assistant needs to be corrected), guiding the assistant through complex reasoning to either **successfully complete the task** or **gracefully handle the inability to do so**. The end result should read like a genuine interaction between a user and a smart assistant with access to certain apps (calendar, email, drive, maps, etc.), while also serving as a high-quality training example.

Key objectives in AC tasks:

- **Showcase the assistant's reasoning and tool use** across multiple turns (not one-and-done answers).
- **Maintain natural, engaging dialogue** (the user sounds human, and the assistant is helpful and clear).
- **Use tools correctly and effectively** (the assistant only calls permitted tools, with proper parameters, and only when needed).
- **Adhere to all guidelines and formatting** to ensure the conversation meets quality standards (this playbook covers these in detail).

Task types: AC tasks generally fall into three categories:

- **Feasible tasks:** The user's request *can* be fulfilled with the available tools. The assistant should solve it using those tools (e.g., finding info in Google Drive, scheduling a meeting).
- **Infeasible tasks:** The user's request *cannot* be fully solved with the given tools. The assistant should make a good-faith attempt and then explain why it can't be completed (e.g., user asks for something outside the tool's scope).
- **General chat:** A conversation not primarily requiring tools (e.g., a casual question or a request that the assistant can handle without the provided tools). These are included to ensure the assistant isn't overly reliant on tools when they are inappropriate.

The following sections will guide you through designing each aspect of the conversation: from crafting the user prompts, to structuring the assistant's reasoning and tool calls, to formatting the final answers and applying error checks.

For this phase, we will not have a **General chat** category.

Checklist:

1. At least 1 Infeasible Turn
2. Complex SM Instructions
3. At least 1 turn having parallel tool calls
4. Sequential tool calls
5. New tools usage
6. Usage of non-required (optional) parameters in the tools
7. 3 Model Failures
8. datetime-reasoning (if applicable)
9. Respecting the task category
10. Conversation length: 10-13/14

Task Categories and Scenarios

When creating a scenario, first determine which category it falls into and design accordingly:

- **Feasible Tool Use Tasks: The user's request can be satisfied using the available tools.**
    - ***Goal:*** Guide the assistant to complete the task using one or more tools.
    - ***Approach:*** The user should present a query or problem that will require the assistant to use the tools to find information or perform an action. The assistant will then plan a solution and invoke the appropriate functions.
    - In feasible tasks, the conversation should usually end with the user's request fully resolved.

- **Infeasible Tool Use Tasks: The user asks for something that the available tools can't achieve (either because the data isn't there or it's out of scope).**
    - ***Goal:*** Demonstrate the assistant making reasonable attempts with the tools and then **failing gracefully**. The assistant should clearly explain the situation to the user.
    - *Approach:* The user's request might seem related to the tool domain, so it's natural for the assistant to try. The assistant will use one or more tools in an attempt to find or do what was asked. When those attempts yield no result or it becomes clear the request is impossible, the assistant explains why it cannot be completed.
    - In infeasible tasks, **no imaginary data** should be produced – the assistant shouldn't make up an answer. It should report the attempts made and the reason for failure (e.g., "no records found" or "this information isn't available via our tools").
    - **Not all the turns should be infeasible; 1 or 2 infeasible turns are enough, and the rest of the turns can be feasible.**

Note: For this phase we are only focusing on Infeasible Tool Use Tasks.

task_category: 80% feasible turns, 20% Infeasible turns.

- **General Chat Tasks: A user query that does not require the specialized tools, often answerable by general knowledge or simple reasoning.**
    - ***Goal:*** Show that the assistant can handle a normal question or request without unnecessarily invoking tools, while still keeping the conversation within a realistic context.
    - ***Approach:*** The user's prompts here are usually straightforward questions or small talk. The assistant should respond helpfully using its general knowledge (since the tools are not needed or helpful for this request). *However*, even general chats should **stay within the realm of the assistant's capabilities and the scenario's context** – avoid completely unrelated trivia that has nothing to do with the tools, if possible. The key is to avoid forcing tool use where it doesn't make sense.
    - ***Example:*** *"How can I travel from India to Paris for my honeymoon?"* – This becomes a general knowledge query. The assistant might respond with a generic answer.
    - General chats ensure the assistant won't try to awkwardly use tools for every single request. They should be used sparingly and still feel like part of the overall suite of tasks (for instance, a casual question a user might ask).

Task Sub-Categories and Scenarios

When creating a scenario, first determine which category it falls into and design accordingly:

- ***Lazy User:***
    - ***Goal:*** Simulate a user who provides minimal input, forcing the assistant to ask follow-up questions and confirm assumptions. The user only supplies one piece of information at a time and corrects the assistant if incorrect assumptions are made.
    - ***Approach:***
        - Design the user to behave passively.
        - Provide vague, underspecified prompts initially.
        - Only provide 1 detail at a time when asked.
        - Introduce corrections if the assistant makes an assumption or over-responds.
        - Example

Note: We can have complex user prompts by maintaining a lazy user tone for complex tool call chaining.

- ***State Dependency:***
    - ***Goal:*** Simulate realistic tool failures that result from system/device state constraints (e.g., location services disabled, Wi-Fi blocked due to battery-saving mode). These tasks evaluate whether the assistant can detect, respond, and recover gracefully based on dynamic tool feedback and System Message logic.
    - ***Approach:***
        - Design scenarios where tool calls fail due to current system states.
        - The assistant should **fetch current settings**, **infer what change is needed**, and **respond appropriately** based on the **System Message policy**.
        - Recovery can include:
            - Automatically changing the state.
            - Asking the user for permission (if required by system message).
            - Suggesting a workaround.
        - Example

- ***Error Recovery:***
  - ***Goal:*** Train and evaluate the assistant's ability to recover from its own tool invocation mistakes. These tasks simulate realistic failure modes caused by misjudged parameters, missing information, or overly constrained tool usage — and then observe whether the assistant can recover naturally in follow-up turns.
  - ***Approach:***
    - **User prompt** triggers assistant reasoning.
    - **Assistant decides tool and argument set** based on available context.
    - Assistant proceeds with the tool call — but:
      - May **miss required arguments** (e.g., providing only `lowerbound` when both `lowerbound` and `upperbound` are required)
      - May **include unnecessary or overly restrictive filters** (e.g., too specific `title` that doesn't match actual data)
      - Or may **violate tool constraints or preconditions** (e.g., providing arguments in an unsupported combination or format)
    - **If the tool fails** (returns error), **do NOT penalize or "fix"** it immediately.
    - Let the assistant **retry at least once**:
      - It might broaden the query.
      - Remove filters.
      - Ask the user for clarification.
    - **Only if the assistant fails again**, the user should then step in to **nudge the assistant** by doing correct tool call considering:
      - Suggesting a simpler/fallback approach.
      - Giving more context.
      - Correcting earlier assumptions.
    - The assistant should **then recover and successfully fulfill the request**.
    - Example

- ***Task Switching:***
  - ***Goal:*** Simulate natural shifts in user attention by introducing temporary topic changes (task detours) mid-conversation. The user interrupts an ongoing task, switches context (e.g., asks about settings or preferences), and returns to the original task once the detour is complete.
  - ***Approach:***
    - **Start a conversation focused on a primary task (e.g., scheduling an event).**
    - **Midway, the user should ask an unrelated question (e.g., system setting, location, past schedule).**
    - **The assistant should gracefully pause the main task, address the detour, and then resume the main task logically and contextually.**
    - **When resuming, assistant must retain earlier context and not re-ask what's already confirmed.**
    - ❌ **Do not fully abandon the first task after the detour.**
    - Example

- [round_7_update] ***Natural User:***
    - ***Goal:*** Simulate a user who is naturally conversational and somewhat informative, unlike the minimalistic Lazy User. Prompts should feel realistic and casual but still lead to multi-step reasoning and tool chaining by the assistant.
    - ***Approach:***
        - Users should speak in full, natural-sounding sentences (not terse or robotic).
        - Each prompt should **imply multiple steps** or require **chained tool use**, even if the user doesn't state all intermediate details.
        - Avoid over-specifying or frontloading all parameters in a single prompt.
        - Ensure at least **3 such user turns** exist within a single conversation trajectory.
    - **Key Design Patterns:**
        - Encourage prompts like:
            - "How many tracks are in *Human After All*?" ⟶ Requires:
                1. Search album
                2. Get album ID
                3. Fetch track details
            - "Directions to the closest McDonald's." ⟶ Requires:
                1. Get current location
                2. Search nearby McDonald's
                3. Get directions to the closest one

These examples show natural prompts that **implicitly expect multiple tool calls** for fulfillment.

Important Notes:

- This is distinct from Lazy User because the user is **not passive or vague**—they're just realistically casual and brief.
- Do **not** artificially break user intent across multiple turns. Let the **assistant drive the breakdown** via reasoning and tool planning.

Creating System Prompt

You should create system prompts that define the Agent's expected behavior, under the following categories. System prompt for each annotation datapoint must be unique.

Good Example: https://simonwillison.net/2025/May/25/claude-4-system-prompt/

Context Information

You can provide identity prompts to define the agent's identity.

For example:

```
None
You are a helpful agent developed by Human corporation. You
name is IRIS. For any questions related to your design and
origin, direct the user to www.human_corporation.com/iris

...
```

Sometimes there are tools that provide access to context information, like the current location, current time, device settings etc. For example:

```
None
Location: Montalvo Arts Center, Saratoga, California, USA
(37°14'26"N 122°1'49"W)

Current time: Friday, May 23, 2025 at 4:08 PM

Wifi: off

...
```

- You can choose to provide all context information, a subset, a superset or no context information in your system prompt. E.g.

```None
You are a virtual assistant on a personal user device. You
are currently located in 5000 Forbes Ave, Pittsburgh, PA
15213.

The user's phone number is 412-302-1232.

Here on the list of Apps you have access to:

Calendar

Settings

...
```

- If a context information is provided, e.g. current location, the agent should no longer need to invoke related tool to acquire context information.

- If a context information is modified and its status is unclear, the agent would need to invoke the related tool instead of relying on stale system prompt context information.
- [round_7_update] You can provide context information about applications and entities the user is currently working with.
  - This info is immediately available to the model, and search won't be necessary when referring to this info.
  - The user could refer to this info simply with "this", requiring the model to refer to this context.

```None
The user is currently looking at spotify app. The song currently playing
is
    uri:"spotify:track:5q2KGgJV5l4MAnnGIrvKSK"
    id:"5q2KGgJV5l4MAnnGIrvKSK"
    name:"Human After All / Together / One More Time / Music Sounds
Better with You"
    uri:"spotify:artist:4tZwfgrHOc3mvqYlEYSvVi"
    name:"Daft Punk"
The page the user is looking at contains
    uri:"spotify:artist:4tZwfgrHOc3mvqYlEYSvVi"
    name:"Daft Punk"
    uri:"spotify:artist:1gR0gsQYfi6joyO1dlp76N"
    name:"Justice"
    uri:"spotify:artist:3AA28KZvwAUcZuOKwyblJQ"
    name:"Gorillaz"
    uri:"spotify:artist:066X20Nz7iquqkkCW6Jxy6"
    name:"LCD Soundsystem"
```

Tool Invocation Instruction

You can offer certain requirements for how tools should be invoked.

- You can define certain actions to always happen:

```None
# Tool Use Policy



## Settings

- When modifying settings database, make sure you confirm
with the user beforehand.

- Some tools require certain device settings to be enabled
in order to be executed (e.g. accessing the web requires
wifi). Always check if those device settings are as expected
before calling these tools.
```

- Make the instructions very fine-granular:

```None
# Tool Use Policy
```

```
## Settings

- When modifying settings database, make sure you confirm
with the user beforehand.

- Some tools require certain device settings to be enabled
in order to be executed (e.g. accessing the web requires
wifi). Always check if those device settings are as expected
before calling these tools. If the settings were checked
before and have not been modified since, you don't need to
check again.

## Calendar

- When creating calendar events, make sure to confirm with
the user which calendar they want to set it in. However, if
an event obviously belongs to an existing calendar of an
event that was just searched / created, no need to confirm.
```

- You can explain the caveats of certain tools:

```None
A couple things to keep in mind especially for calendar
events:

1. When searching for calendar events, always provide a time
range, otherwise you won't be able to locate recurring
events that happen indefinitely.

2. If a recurring event is found, the search result will
also return the parent recurrence of that event to let you
know the info about the recurrence like its interval.

3. If you want to delete / modify a recurring event
instance, use the instance ID. If you want to delete all
instances, use the parent ID. Always confirm with the user
which one they want in face of this.
```

- You can explain the inner working of certain tools:

```None
Here's what each internal database field of settings means:

- `device_id`: UUID

- `place_id`, `latitude`, `longitude`, and
`formatted_address`: Canonical information of the current
location. They must be consistent.

  Can be found through Google Search, as well as
https://developers.google.com/maps/documentation/places/web-
service/place-id.

- `wifi`: When disabled, wifi-related tools will result in
errors (e.g. searching maps), until wifi is turned on.
```

```
- `location_service`: When disabled,
location_service-related tools will result in errors (e.g.
getting current location info), until location_service is
turned on.

- `cellular`: When disabled, cellular-related tools will
result in errors (e.g. sending messages), until cellular is
turned on.

- `low_battery_mode`: When enabled, `wifi`,
`location_service`, and `cellular` must be turned off. When
enabled, `wifi`, `location_service`, and `cellular` cannot
be turned on until `low_battery_mode` is turned off.

- `utc_offset_seconds`: Number of seconds the current device
timezone differs from UTC. E.g. -25200.0 represents
UTC-07:00

- `locale`: Locale code. This will affect the default
behavior of locale-sensitive tools when locale arguments are
not provided with the request (e.g. searching maps).
```

- When there are overlapping tool functionalities, you can ask the model to never use some tools in cases, prefer one over the other in certain cases, use any of them, or use both for cross-referencing:

```None
- When the user asks for something you can answer with your
own knowledge, never use web_search.

- If the user asks for restaurant related information,
prefer to use yelp over google maps.

- If the user asks for something you can answer with other
tools, prefer them over web_search, excpt for e-commerce,
see the next bullet for more details.
```

```
- If the user asks for shopping advices, cross reference
reviews between amazon, reddit and web_search to summarize
your findings.
```

- You can control when parallel tool calls should be issued:

```None
When calling place_details, you can issue multiple tool
calls in parallel. For create_calendar_event, always issue
them one at a time.
```

User Preference

You can explain certain user preferences that can help the agent make better decisions.

- You can add user preference among entities:

```None
The user likes Thai food the most. The user likes to refer
to John Petrucci as “my guy.”
```

- You can add engagement preferences:

```None
Note: The user really hates it when you ask too many
questions. Make sure to solve the problem yourself as much
as possible, even when there's a chance it could be wrong.
```

- You can add background information:

```None
The user just got back from a trip from NYC and is very
sleep-deprived. He likely won't appreciate early meetings
and could really use some light food.
```

Tonal Control

You can control how the model responds to the user.

```None
You should assume that the user does not have access to
anything visually. You should explain yourself in detail in
a colloquial tone, as if you are directly speaking to the
user. Don't be too formal. Be lighthearted and cheerful.
```

```None
###### PAY SPECIAL ATTENTION ########


The user is able to see the screen while you are talking.
You don't need to explain every detail. Provide the key
information for the user to confirm what you did.
Proactively suggest if the user would like to make changes
or not.
```

Wrapping Up

In the end, you should have a system prompt that combines multiple categories above in a coherent manner.
 The system prompt should mix and match requirements from each category in a random order. Each system prompt should be unique and **should not follow the same formatting or structure**.

Do not make it obvious that the prompt is broken down by these categories.

Agent behavior will be judged based on the system prompt.

Example:

```None
####### Speaking to the user #######

| Talk to the user as if you are speaking to them directly. Make sure to
be natural and concise, don't take up too much time, but convey all
necessary information.

| Make sure information is correct and factual especially when providing
confirmation or disambiguation.

| Act as if you are a professional human assistant, with a somewhat
jestful attitude. Throw in a couple of jokes here and there won't hurt.

| The user is from NYC and he is particularly picky about pizza. Be
mindful about this when selecting pizza restaurants.

####### Calling Tools #######

| Even though you have access to settings-related tools, do not invoke
them no matter what. Suggest the user modify them themselves. The user
really hates their settings being messed around with.

| Always check if necessary setting is enabled when calling certain tools
just to be safe.

| You don't need to confirm with the user when creating or modifying a
calendar event, but make sure to always confirm when deleting one.

| When creating a calendar event with location information, make sure to
fill in everything including address, lat, lng, and place_id.

| Before creating a calendar event, make sure to search first to check if
there's any conflict. If so, prompt the user what to do.

####### Other Info #######

| You are in Montalvo Arts Center, Saratoga, California, USA (37°14'26"N
122°1'49"W), but that might change in a bit. After a few rounds of
interactions, if you are unsure you should check again.

| Time is always expressed with ISO datetime string, including UTC
offset.
```

Points to be considered:

The following advanced guidelines should be followed when crafting System Prompts for agent behavior. These go beyond the earlier structure and aim to improve variability, realism, and instruction-following.

---

1. Vary Formatting Structure

- Avoid rigid markdown headers and templated categories (e.g., Context, Tools, Tone).
- Instead, blend behavioral instructions naturally into prose or bullet formats.
- Use diverse structural styles—dialogue, lists, embedded constraints in narrative, or informal notes.
- Don't use the same format across prompts; visual layout should feel unique each time.

---

2. Increase Tonal Variability

- Use broader tone directives beyond “formal/informal”:
    - Examples: “Keep it breezy like you’re chatting with a friend,” “Be brief and curt, like a busy coworker,” “Sound like a game show host,” “Use poetic metaphors,” etc.
- Mix verbosity levels:
    - Some prompts should encourage terse, quick responses; others, detailed and explanatory.
- Introduce roleplay or stylized personas:
    - E.g., “You’re an enthusiastic travel guide,” or “You speak like a sarcastic AI mentor.”

---

3. Ensure Instruction-Tone Alignment

- The assistant must follow tone instructions given in the system prompt precisely:
    - If told to be casual and brief, it must not produce long formal paragraphs.
    - If the prompt says "Inform after changing the permission…", the assistant must inform the user afterward.
- Annotators should monitor whether assistant behavior truly aligns with tone and execution cues in the system prompt.

---

4. Enrich Contextual Information

- Include richer and more varied contextual cues:
    - Examples: Current location, local time/date, device battery status, locale, list of installed apps, recent user actions.
- Some prompts should intentionally omit context, requiring the model to use tools to retrieve it.

---

5. Decouple Prompt from Task

- Avoid system prompts that perfectly align with the user task—this causes overfitting.
- System prompt should contain mixed relevance:
    - Some info should be clearly helpful, some irrelevant, and some misleading unless validated.
- This encourages the model to reason rather than blindly obey prompt details.

---

6. Diversify Tool Invocation Guidelines

- Tool-related instructions should not repeat the same phrasing across prompts.
    - Avoid repeated use of:
        - “Never assume current location”
        - “Don’t say please/thank you”
        - “Always confirm calendar name”
- Vary the specificity, caveats, and even priority of tool rules:
    - Example: One prompt might say “Always check Wi-Fi before maps,” another might say “Skip Wi-Fi check if it was just enabled.”
- Include conditional logic in prompts to test adaptability:
    - E.g., “Enable cellular without asking, unless low battery mode is on.”

Conversation Structure

Agent Completion dialogues are meant to illustrate the assistant's reasoning. Rather than the assistant immediately producing an answer, it should **think through the problem step by step**, often calling functions along the way. Here's how to structure the conversation:

- **Turn Definition:** A "turn" consists of one user message, the assistant's response *and optionally one or more tool function calls* (with their outputs) triggered by that assistant response. In practice, the sequence looks like:
  - **User message.** (Always starts a turn.)
  - **Tool call(s) and output.** (Optional, depending on whether the assistant needed a tool for that turn.) The assistant's message triggers a tool, which returns some output. The output is shown.
  - **Assistant continues...** Sometimes the assistant might immediately continue after a tool output with more reasoning or another tool call or asking for more details to continue.
- In our context, we often break a single turn with multiple function calls into separate steps for clarity. However, it's important to maintain the correct message order: **user**→ **tool call** → **tool output** → **assistant** (and repeat).
- **Multiple turns:** Aim to have **multiple reasoning turns** in one conversation. Rather than solving everything in one giant assistant answer with many calls, it's more realistic to have a back-and-forth:
  - The user asks something.
  - The assistant responds with tool calls → gets results.
  - The user might clarify or ask a follow-up based on those results.
  - The assistant responds again with further tools or final info.
  - (And so on…)

    OR
  - The user asks something.
  - The assistant requests more details.
  - The user might clarify.
  - The assistant responds again with further tools or final info.
  - (And so on…)
- **No redundant or dummy calls:** Every tool invocation should have a clear and necessary purpose. Avoid calling functions merely to meet a quota. For instance, if information like the current location has already been obtained through a previous call, there's no need to call the function again—reuse the existing data. Unnecessary or redundant calls not only reduce efficiency but are also flagged as errors (see `tool_over_triggered` in Error Labels).

- **Sequential vs Parallel calls within a turn:** If the assistant needs multiple pieces of information that are independent, it can call functions in parallel (i.e., list multiple function calls one after the other in the same assistant message before waiting for outputs). For instance, searching two different keywords in Gmail at once using two `gmail_search` calls in parallel is fine. However, if a subsequent call depends on a previous call's result, those must be sequential (one after the other, with the output used in the next step). For example, the assistant must **first** use `search_place` to get a place ID, **then** use `get_directions` with that place ID. It shouldn't do both in parallel because the second needs data from the first.

- **Logical interdependence between turns:** Keep a strong thread through the conversation. The result of turn 1 should inform turn 2, etc. For example, the user asks for a document, the assistant finds a list of documents in Turn 1. The user then says "open the second one", the assistant does that in Turn 2. This kind of inter-turn dependency makes the conversation coherent and realistic. Avoid turns that feel disconnected or ignore what happened before.

- **Summarizing progress:** After a series of tool calls (especially parallel calls or a sequence of calls retrieving data), it's good for the assistant to **summarize what it found before moving on** or before asking the user a follow-up. For instance, if the assistant just fetched three emails with `gmail_search`, it might summarize: "I found 3 emails about Project X. One is a status update, another is a budget, and the third is a meeting invite." Then it might ask, "Would you like a summary of one of these, or should I open the budget file?" Such summaries serve two purposes: they inform the user, and they make sure the conversation captures what the assistant "knows" so far in a readable way.

- **Ending conditions:** A conversation can end when the user's request is fulfilled or when the user doesn't need more. Make sure the conversation doesn't cut off abruptly. Usually, the user gets their answer or an apology for not succeeding, make sure it's not ending at the user level, the last turn should always be the assistant's response.

Remember, while we simulate both roles, **the user side should feel real** – sometimes impatient, sometimes asking for clarity, sometimes providing new info, etc., and **the assistant side should reflect a thoughtful AI agent** – analyzing, using tools, and explaining. This dynamic structure is what makes the dataset valuable.

User Prompt Design (Natural & Varied)

The **user prompts** (the things the user says) drive the whole conversation. They must be written in a natural, **human-like style**, not as if we are scripting the assistant's behavior. Here's how to craft user utterances:

- **Chain-of-Tool-Use Encouragement:** As a general directional guideline, we are encouraged to design **more organic user prompts** that **naturally trigger multiple tool calls** within the same turn. This means moving away from overly structured or atomic interactions (e.g., *"User asks → one tool call → assistant responds"*) and toward **richer prompts** that lead to **sequential or parallel tool invocations**.
    - **You may slightly relax the "Lazy User" constraint** to enable such prompts when needed.
    - There is **no strict quota** for multi-tool examples yet, but we aim to **gradually shift the dataset** to reflect **more realistic, complex assistant behavior**.
    - Examples should reflect:
        - User asking for a plan or output that inherently needs chaining tools.
        - Natural language that implies multiple steps (e.g., searching, confirming, then creating).
        - Situations where the assistant has to fetch supporting info before acting.
    - This improves the realism, complexity, and training signal of our conversations.

- **Datetime Reasoning (When applicable):** When creating user queries that requires grounding relative / absolute datetime to tool arguments, e.g. Calendar date time, direction departure time, make sure to create challenging datetime reasoning tasks, including but not limited to:
    - Unit Conversion: "Mark something on my calendar in 120s", should be converted to 2 minutes if the tool requires ISO 8601
    - Complex Relative Time Calculation: "What's on my calendar in 40 days / the last Friday next month", need to calculate month and day based on current datetime and delta
    - Natural Language Understanding: "How long is my commute if I leave half past noon tomorrow"
    - Fuzzy Datetime Range: "Do I have anything scheduled tomorrow at brunch time", either use a somewhat large range based on common sense, or confirm with the user.

- [round_7_update] **Search Refinement:** In this category, you need to create queries that would require the model to gradually refine its search queries to find the relevant information.

    - Search related tools are sometimes not accurate enough, which requires the model to modify arguments or try other tools.

    - Examples:
        - When the user asks to search for ‘Dream Theater concert’, search_events, if no relevant results are seen, spotify_artist_goods, then web_search, then search_calendar_events.
        - When the user asked to search for meeting with John in the morning, the model might start by searching for “John” as query and 9AM - 12AM as starting time range, if no relevant results, try using “John” as participant name and 9AM - 12AM, if no relevant results, try 9AM - 12AM, and in the end, try searching for the whole day.
    - You can control how far and wide you’d like search refinement to go, through the system prompt as well.

- **Sounds natural, not scripted for tools:** The user should **never reference the tools or functions by name**. Avoid writing a user message that a real person wouldn’t say. For example, a user is unlikely to ask: *“Can you use the* `convert_place_id_lat_lon_address` *function to get coordinates for 1250 Industrial Parkway?”* – that is artificial. The assistant’s job is to figure out which function it should use. Always phrase prompts in terms of the user’s problem or request, not in terms of what the assistant should do internally.

- **Provide context but not too much persona:** It’s okay to set a brief context in the first user turn (like *“Hi, I’m planning a team event and need some help.”*). This can ground the conversation. However, **avoid long personal backstories or extraneous details** in the first prompt. The advice is that a persona or context is **“not needed in the first prompt,”** meaning you can often dive right into the request. If you do include context, keep it concise and relevant.

- **Avoid unnatural specifics:** Don’t include information a user wouldn’t normally provide. For example, users don’t typically give exact latitude/longitude or database IDs – they expect the assistant or system to handle that. If your prompt has something oddly specific (like a long ID or coordinate) just to force a tool to be used, reconsider. The challenge should be for the assistant to acquire that info via tools, not the user feeding it. **Example:** Instead of *“I’m at 40.7306, -73.9352, find a café near me,”* a user would say, *“I’m near East Village in NYC, can you find a café to study?”* – then the assistant might use coordinates internally.

- **No explicit “thanks” or sign-offs in the middle:** Users usually don’t say “thank you” after every assistant response in the middle of a task – they wait until the end when the task is done. Also, since we simulate multi-turn, the user shouldn’t be excessively polite every single turn (that feels scripted). So, *don’t include “Thanks” or “please” in every user message*.

- **Vary how users ask things:** Not every user prompt should be a question. Real users mix statements, commands, and questions. For example:
    - "Show me the latest report for Q3" (command-like).
    - "Can you check if I have any meetings this afternoon?" (question).
    - "I'm not sure what's on my schedule next week, plz help me with that." (statement implying a request).
    - "Find me some trails in Rocky Mountain National Park." (imperative).
    - Followed by: "Which one has the best alpine scenery?" (question referring to prior result).
- This variation makes the dialogue dynamic. Avoid having all prompts structured like "Can you…?" repeatedly. It's noted that using *different wording for questions* (not always starting with "Can…") is important.
- **No "Provide me with …"**: Phrasing like "Provide me with details about X" or "Provide a route from Y to Z" is discouraged for user prompts. It sounds like an unnatural command a developer might give. Instead, rephrase as the way a user would naturally speak:
    - ❌ *"Provide me with details about Yekatit 12 Hospital."*
    - ✅ *"Tell me more about Yekatit 12 Hospital."*

    - ❌ *"Provide me with a walking route from my current location."*
    - ✅ *"How do I get there on foot?"* (assuming context of "there" is established).
- **But don't reveal the exact answer in the prompt:** The user shouldn't hand the assistant everything. For instance, the user shouldn't say, "There's a file called `Budget_Q4.xlsx`. Find it." – that's too easy/direct. Instead, a realistic user would say, "I need the Q4 budget file" and possibly who authored it or where it might be, and the assistant has to figure out which file that is via search.
- **Avoid repetitive patterns across conversations:** If you're creating multiple tasks, ensure each user scenario feels distinct. Don't have every user start with "Hi, I'm working on X, I need Y." Change it up: different personas (one might be a marketing manager, another a software engineer, another a student), different styles (one very formal, one very casual, one very urgent). This diversity is important so the model doesn't overfit to one style.
- **Follow-up prompts refer back appropriately:** When the user references something from earlier, they should use pronouns or shorthand naturally. For example, user's first prompt: "Find me some Rocky Mountain National Park trails." The assistant gives results.

Next user prompt: “Which one has the best alpine plants?” – the user says “which one,” assuming the assistant knows it refers to the trails. This is good; it forces the assistant to use context. The alternative, repeating “Which trail has the best alpine plants?” would be okay, but using “which one” is more conversational. Similarly, a user might say “How do I get there?”, following that – meaning directions to that trail. This kind of ellipsis (there, it, they) should be used when it makes sense, rather than the user re-stating everything.

- **No tool-specific jargon from user:** If the user wants directions, they’ll say “How do I get to X?” not “Call the directions API” or even “get me a route” (the word “route” is fine in user language, but something like “provide navigation” is a bit formal – still okay, but just consider what’s natural). If the user wants to search their emails, they’d just ask about the content of the email (“Did I receive an email about the budget report?”) rather than “use the Gmail search function”.

Good vs. Bad Examples:

Note: For `Lazy User,` simulate a user who provides minimal input, forcing the assistant to ask follow-up questions and confirm assumptions. The user only supplies one piece of information at a time and corrects the assistant if incorrect assumptions are made.

Good Prompts:

1. Show me nearby vegetarian pizza places rated above 4.5.
   Why it's good:
   - Combines filtering and reasoning.
   - Clearly defines search intent (vegetarian, rating-based).
   - No internal knowledge or tool reference.
   - Natural phrasing; doesn't sound robotic.
2. Can you check if I'm free for 2 hours next Thursday after noon and block that time for a brainstorming session with Rashmi?
   Why it's good:
   - Leads naturally to calendar tool usage (create + search).
   - Sounds like a real human task.
   - Involves a named participant and purpose for the event.
3. Find a bookstore nearby that's known for a quiet reading section and doesn't close before 8 PM.
   Why it's good:
   - Uses human language instead of system terms.
   - Combines place type, ambiance, and operating hours—good filtering.
   - Sounds like something a person would say naturally.

Bad Prompts:

1. I'm a process engineer at Global Glass Manufacturing and need coordinates of 1250 Industrial Parkway.
   Why it's bad:
   - Persona inclusion and unnecessary backstory.
   - Artificially engineered to trigger a specific tool (coordinates).
   - Doesn't match natural user behavior.

2. I'm looking for a quiet place to study near latitude 40.7306 and longitude -73.9352 in New York. Can you help me find a study-friendly café with a good ambiance for focused reading?
   Why it’s bad:
   - Use of Latitude/Longitude Coordinates
   - Redundant Context Specification
   - Implies Tool Awareness
3. I just landed at San Francisco, please get_direction to Google office, make sure to have traffic to be pessimistic and mode to be transit.
   Why it’s bad:
   - Users don’t know or say internal tool name
   - users say “avoid traffic” or “use public transport,” not tool parameters.
   - Doesn't match natural user behavior.

Example of transforming an unnatural prompt into a natural one:

1. **Unrealistic Prompt:**
   “Provide me with details about Yekatit 12 Hospital.”
   Realistic Prompt:
   “Tell me more about Yekatit 12 Hospital.”

2. **Unrealistic Prompt:**
   “Provide me with a walking route from my current location.”
   Realistic Prompt:
   User: “How can I get there on foot from here?”

3. **Unrealistic Prompt:**
   “Hi, I’m an architecture student working on my thesis about quiet public spaces, and I’m looking to spend the evening in a bookstore within exactly a 3,000-meter radius of my current geolocation. It’s important that the place allows extended stays till 8PM evening, has reliable power outlets for device charging, and provides a serene reading environment with minimal background noise.”
   Realistic Prompt:
   User: “How can I get there on foot from here?Can you find bookstores within 3 km that have a reading area, offer device charging, and close by 8 PM?”

4. **Unrealistic Prompt:**
   “I’m considering going out for dinner tomorrow evening, so first, identify a well-rated italian restaurant within close proximity to my current location. Then, please verify my availability in that time window by reviewing my personal calendar. If there are no existing events or conflicts, proceed to reserve that time slot for dinner and finally provide me with navigational instructions to reach the top rated restaurant, you identified.”
   Realistic Prompt:
   User: “Check if I’m free tomorrow evening. If I am, then block that time for dinner in my calendar, also find a nearby top rated italian restaurant, and send me directions to it.”

By following these guidelines, the user side of your conversation will feel authentic. This challenges the assistant to respond in a natural, intelligent manner, which is our goal.

Assistant Reasoning & Tone

Note: This behavior might differ if we mention something in SM, so it should follow what is mentioned in the SM.

The assistant's role is to be intelligent, helpful, and clear. In AC tasks, the assistant also needs to **show its thought process** and use of tools, which is a bit artificial in the sense that a real assistant wouldn't spell out "I will do X." However, we want the assistant's reasoning to be written in a way that *feels natural and helpful*, not robotic or overly formal. Here's how to write the assistant's messages:

- **Take initiative and lead the process:** The assistant should never ask the user for permission to do something that the user's request implicitly asks for. For example, if the user asks "Find X for me," the assistant should not respond with "Should I search for X in your Drive?" – it should just do it. The user has already implicitly given the go-ahead by asking. This doesn't mean the assistant never asks questions – it *should ask* the user if something is unclear (see clarification below) – but it shouldn't be timid. The tone is confident and action-oriented. Think of it like a knowledgeable assistant who knows its tools and uses them without being told explicitly.

- **Ask for missing information (clarification):** If the user's request is incomplete or ambiguous in a critical way, the assistant **must ask** rather than assume. For example, user says: "Schedule a meeting with Alex on Monday." If the assistant has multiple Alex contacts or multiple calendars, it should reply: "Sure. Which Alex do you mean (Alex Johnson or Alex Lee)? And on which calendar should I create this event?" This is far better than guessing the wrong Alex or defaulting to an arbitrary calendar. However, **do not over-ask**: if something is likely or the user gave a hint, you can make a reasonable assumption *and confirm it*. E.g., "I'll assume you mean Alex Johnson since he's in your contacts. Let me know if that's not correct." This shows initiative but still allows the user to correct.

- **Maintain a professional but friendly tone:** The assistant should be polite and use complete sentences, but not overly flowery. Avoid overly formal language like "As per your request, I shall now proceed to execute a search…" – that's too stiff. Also, avoid too much slang or casualness unless the user's tone clearly invites it. A good middle ground: "Alright, I'll search for that document in your Drive now." – Friendly, first-person, contractions are fine (I'll vs. I will) since that sounds normal in conversation.

- **No over-personalization or chit-chat:** The assistant shouldn't insert opinions about itself or unnecessary personal comments. For example, avoid "I'm happy to help!" or "I think this is interesting," unless the user said something that warrants an empathetic response. Stick to the task. Also, the assistant doesn't express frustration or get flustered; it's patient and steady. "Over-personalization" (the assistant talking about its feelings or excessively using “I” for no reason) is marked down. Small polite phrases like “Let me see what I can find” are fine – they express a bit of personality without going off track.

- **Use of bullet points and formatting for clarity:** When the assistant has a lot of information to present (like multiple search results, or a summary of a document, or step-by-step instructions), it’s often best to format it in a clear way:
  - Use **bullet points or numbered lists** to enumerate items. e.g., “I found the following files:\n1. `ProjectX_Plan.docx` – last edited 5 Jan 2025\n2. `ProjectX_Budget.xlsx` – last edited 10 Dec 2024”.
  - Use **bold** to highlight field names or important words, especially in summaries. For instance: **Title:** Quarterly Budget Meeting – **Date:** Jan 5, 2025 – **Attendees:** John, Alice.
  - The conversation output supports Markdown, so leverage that. Just ensure the formatting is correct (for example, after a bold label like `**Date:**` include a space before the value, so it renders correctly as "**Date:** value").

- **Keep responses focused:** The assistant should answer the user’s question or address the request **directly and thoroughly, but without going on tangents**. For example, if the user asks to summarize a document, the assistant should summarize it and not throw in unrelated advice or alternate topics. Stay on task, and once the request is fulfilled (or cannot be fulfilled), look to wrap up that thread of conversation or ask if the user needs anything else related.

- **Summarizing and concluding:** After the assistant has gathered info via tools, its response should summarize **in human terms**:

    - Don't just paste raw JSON or verbose outputs (except what the platform automatically shows from the tool). Instead, the assistant picks out the relevant pieces and explains them.
    - Example: The tool returns an event with a long description, attendees list, etc. The assistant can say: "There's an event titled 'Team Sync' on **March 10, 2025, at 3:00 PM** in **Work Calendar**. It's located at **Conference Room B** and the attendees are you (organizer), Alice, and Bob. The description mentions it's a quarterly update meeting." This is much nicer than just dumping the JSON fields.
    - The assistant should ensure **all important details** from the tool output that the user would care about are included. Missing a key detail (like not mentioning the location of a meeting) would be an oversight.

- **Graceful failure messaging:** If in an infeasible scenario or a tool error:

    - The assistant should clearly explain what it attempted ("I searched our database and the shared drive for 'Project Y contract'...") and then explain the result ("...but I couldn't find any documents related to that. It might not exist or I might not have access to it.").
    - The key is not to leave the user hanging without explanation.

- **Avoiding repetition:** The assistant should avoid repeating large chunks of text unless for emphasis or clarity. For instance, if a user's question is long, the assistant doesn't need to quote it back verbatim. It can paraphrase if needed ("You're asking for..."). Repeating a user's exact words can seem unnatural or padded.

By following these guidelines, the assistant's messages will be comprehensive, helpful, and aligned with expectations. They'll demonstrate a clear thought process and use of the tools while still feeling like part of a natural dialogue.

Tool Usage Guidelines

One of the most critical aspects of AC tasks is the correct use of **tools (functions)**. The assistant has a toolbox of functions it can call to get information or perform actions on behalf of the user. Proper tool use is what separates a high-quality completion from a failing one. Below are guidelines for using tools:

- Refrain from only selecting Maps, Calendar and Settings tools and only using them in all the tasks. Make sure to include and use at least one new domain.
  - In previous rounds, examples heavily relied on **Maps (Place/Navigation), Calendar, and System Settings** tools. To improve dataset diversity and better train the model across all capabilities, the client now **requires broader domain coverage**.
  - What You Must Do:
    - **Include at least one tool from a different (less used) domain** in each task.
    - Don't limit your tool usage to (Maps/Place), (Calendar), (System Settings)
    - Instead, **diversify your tool selections** by incorporating tools from domains like:
      1. **Spotify** (e.g., `search_spotify_podcasts`, `spotify_album_details`)
      2. **Product** (e.g., `product_details`, `product_reviews`)
      3. **News** (e.g., `get_health_news`, `search_news`)
      4. **Stock/Finance** (e.g., `get_market_quotes`, `get_stock_history`)
      5. **Web & Media** (e.g., `web_search`, `search_reddit`)
      6. **Business/Yelp** (e.g., `search_yelp`, `business_reviews`)
  - This change ensures:
    - More varied assistant behavior.
    - Broader tool evaluation and error coverage.
    - Avoids repeating the same patterns as earlier batches.
- Make sure to have sufficient coverage on less used arguments, providing a fairly comprehensive coverage over all tool arguments across the entire data delivery batch.
  - Each tool function has **multiple arguments**.
  - **Issue so far:** Most tasks only use **frequent arguments**, and ignore **less common ones**.
  - **Client expectation:** Across the batch (not necessarily in every task), make sure:
    - You write prompts that **lead the assistant to use** different or uncommon arguments.
    - Tool calls reflect a **diverse and realistic** range of argument usage.

    - **What You Should Do**:
        - Review tool definitions and **list all their arguments**.
        - Create some tasks that intentionally target **less-used arguments**.

- **Use only the allowed tools:** Never invent a new function or call something outside the list provided by the client. Even if the user asks for something that sounds like a different service, you must map it to one of these or explain that it's not possible. *(Using a disallowed tool or an API not in the list is an automatic fail.)*

- **Choose the correct tool for the job:** Many tasks could potentially be addressed by multiple tools, but usually one is the most appropriate.
    - If the user asks something that sounds like needing an external web search (which we might not have), consider if it can be done with available tool (maybe if it's an internal knowledge search) or if not, the assistant might have to say it can't because no web tool is available.
    - **Wrong tool example:** The user asks for directions to a place, and the assistant uses `gmail_search` – that's clearly the wrong tool. Or the user asks to find a file and the assistant uses a calendar function. These would be tagged as `wrong_tool_selected`.

- **No assumption of hidden parameters:** The assistant should not guess things that the user didn't provide if the tool needs it. For example, if searching an email requires a query and the user just said "find that email I got from my bank," you have some info ("from my bank" – likely the sender or subject). Use that. But don't randomly decide the query is, say, `from:noreply@bank.com` unless you have reason. If the user said only "find that email I mentioned," the assistant should ask for more detail. **Never put something in a tool call that wasn't either given by the user or logically deduced from context.** (E.g., don't fabricate an ID or name.)

- **Parallel calls for independent searches:** If the user asks something like "Find any related emails and documents about Project X," the assistant can do two searches in one go:
    - `gmail_search` for "Project X"
    - and `google_drive_search` for "Project X"
- These can be called in parallel within the same assistant turn. This is efficient and demonstrates skill. Just ensure you explain it like: "I will search both your emails and Drive for 'Project X'." Parallel calls are a great way to show the assistant handling multi-faceted queries.

- **Sequential calls for dependent actions:** When one tool's output is needed for another, sequence them properly:

    - E.g., To open a file you found, first do `google_drive_search`, get results, then in the next assistant action (or same turn if you plan it that way) call `open_search_results` with the appropriate file identifier from the search result.
    - Another example: for directions, first use `get_current_location` (if needed) to get user's lat/long, then use `get_directions` with that as a start. This cannot be parallel because you need the location first.
    - **Tip:** In one assistant turn, you could do them sequentially too: call `get_current_location`, get output, then immediately call `get_directions` in the continuation of the assistant's message, and then finally answer. That's fine as long as you described that plan.

- **Respect the conversation ordering with tools:** As mentioned before, the assistant's message triggers the tool, then you get output, then the assistant continues (either in the same turn or the next). Do not put final answers before you have the tool's info.

- **Use multiple tools in a single task:** For extra robust scenarios, you can incorporate different tools in one conversation if it fits the narrative. E.g., a scenario where the user needs to schedule a meeting (Calendar) and then email participants (Gmail draft), or find a document (Drive) and also an email thread about it. This showcases the assistant's versatility. Just ensure the flow is logical (the user asked for those multiple things or one led to the other).

To summarize, **proper tool use** means: correct tool choice, correct parameters, logically integrated into the conversation, and every call is necessary and useful. All of this should happen seamlessly from the user's perspective – the assistant is just being "helpful" – while under the hood we see the tools being used.

Assistant Responses and Summaries

The final answers and how the assistant presents information to the user are the culmination of reasoning and tool use. This section focuses on making sure the assistant's outputs are clear, accurate, and well-formatted.

- **Reflect tool results accurately:** Whenever the assistant provides information that came from a tool, it must be truthful to that tool output. This avoids *hallucination* (making things up) and ensures quality.
  - For example, if the assistant searched an email and the email's snippet says "...the budget is $5,000...", the assistant's summary should not say "the budget is $10,000" or even "thousands of dollars" – it should say $5,000 as in the source (unless the user explicitly asked for rounding or something).
  - If the assistant used multiple tools, it should attribute information correctly. If it mixes them up, that's an error. Keep track: info from calendar vs info from drive should not get conflated.
  - If a tool provided a list of results and the assistant picks one, it should be clear which one it's talking about (use referencing as needed), and there should be a solid reason behind picking that one.

- **No adding unwarranted info:** The assistant has some general knowledge base (the AI's training data), but in these tasks, **we treat the tools as the source of truth** for task-specific info. The assistant shouldn't slip in some extra facts that the user didn't ask and the tools didn't provide. For instance, if summarizing a document, don't add a conclusion that isn't in the doc just because it sounds smart. Stick to what's given or what logically follows from it.
  - An example of what *not* to do: Tool returns an event with no location given, just a title and time. The assistant in summary says "Location is TBD, probably the main office." That "probably" is a guess – avoid that. Just say "Location is not specified."
  - Another: If the user asks a general knowledge question in a general chat and we have no specific tool, the assistant can use its knowledge (since that's allowed in general chat), but if it's a feasible task with available info in tools, it should not rely on its own memory.

- **Confirming actions or critical info:** When the assistant is about to perform a potentially irreversible action (like scheduling a meeting or sending an email) or when it has to make an assumption, it's good to confirm with the user:
  - "Should I go ahead and schedule this meeting at 10 AM Monday on your Work Calendar?" – especially if time or calendar wasn't 100% confirmed.
  - This gives the user a chance to correct if needed, and shows the assistant is careful.
  - However, if the user's request was explicit ("schedule it now"), the assistant can just do it and then tell the user it's done, rather than asking again.
  - Confirm data by repeating it: e.g., "You asked for a list of all files related to Project X. I found 3 files named A, B, C. Do you want details of any specific one?" – here it's confirming understanding and offering next step.
  - When confirming, **use the same terms** the user used. If the user said "my team's calendar", answer with "your team's calendar" (or the actual calendar name if known). Don't introduce new synonyms that might confuse.

- **No raw JSON or overly technical output to user:** The assistant should translate any JSON or code from the tool output into normal language. The user doesn't see the JSON output, but the conversation text should not just be "Here is the data: { ... }". Instead, describe it. The only time we leave something JSON-looking is if the user explicitly asked to see raw data (rare in these tasks) or if maybe an error returns a snippet that we quote. Even then, it's better to explain it in words.

- **If multiple items found, help user navigate them:**
  For instance, if 3 emails were found and the user asked to find “the email from HR about benefits”, and the search returned 3 various HR emails:
  - Assistant might list them:
    1. Email from HR (Jan 5, 2025) – Subject: “Benefits Update 2025”
    2. Email from HR (Dec 20, 2024) – Subject: “Year-end Office Holidays”
    3. Email from HR (Nov 10, 2024) – Subject: “Open Enrollment Reminder”
  - Then the assistant can ask (if the user hasn’t specified which one): “I see a few emails. The first one looks like it’s about benefits updates for 2025. Is that what you’re looking for?” or simply wait if the conversation structure expects the user to choose.
  - The key is, the assistant should not randomly pick one if the user’s intent isn’t certain. Either disambiguate by asking or, if it’s pretty clear, state why you chose one (“I assume the ‘Benefits Update 2025’ is the email you meant since it matches your description.”).

- **Gracefully handle no results:** If a search turned up nothing and it was feasible:
  - Assistant: “I searched your Drive for ‘Project Phoenix summary’ but didn’t find any document with that name or content.” Perhaps follow up: “Maybe it’s stored under a different name? Let me know if you have other details to try.” (This invites the user to clarify).
  - The assistant should be honest about not finding something, and if appropriate, **suggest a logical next step**. But since our conversations are bound to the tools we have, often it ends with apologizing for not finding the info.

- **Final answer completeness:** By the end of the conversation (the final assistant message for that task):
  - If feasible: the user’s goal should be accomplished or the question fully answered. The assistant might say something like “Is there anything else you need?” as a polite close if it fits, but it’s optional.
  - If infeasible: the assistant should have clearly stated it cannot complete the request and possibly the reasons. No loose threads should remain (don’t leave the user expecting something else that never came).
  - If general: the question should be answered or the chat concluded appropriately.
  - Ensure no crucial information is left unsaid. Double-check if the user’s original request had multiple parts, that all were addressed.

By keeping these practices, the assistant’s outputs will not only be correct but also presented in a way that’s easily understood, which is essential for both user satisfaction and for training the model effectively.

Judging Assistant Behavior

Except for issuing accurate tool calls, several other key factors should be taken into consideration when evaluating the quality of assistant response:

- If a **customized system prompt** is provided, one should judge assistant behavior **based on the system prompt instruction**.

- If **no customized prompt** exists, the assistant should default to the behavioral expectations listed below:

Ambiguity handling

If ambiguity exists—e.g., multiple locations are returned by a tool requiring the user to disambiguate, or the user instruction is not clear enough to determine applicable tool/domain—make sure the assistant **requests the user to disambiguate**, instead of making assumptions.

- ✅ Good: "I found multiple places named 'Sunset Grill'. Could you tell me which city you meant?"
- ❌ Bad: "I've included Sunset Grill in Los Angeles." (without confirmation)

Overlapping Tool Functionalities

There could be tools that have overlapping functionalities. By default, we allow the model to use any tool that completes the task. Depending on system prompt instructions, this behavior can be overridden. See `Creating System Prompt` for more information.

Preferring Tools over Memorization

Sometimes it's possible for the assistant to memorize or guess information instead of retrieving it via tools.
To maintain accuracy and freshness, we prefer that the assistant **bias toward tool use**—especially when the tool has authoritative or updated information.

- ✅ Good: Assistant uses `get_system_settings` to check if Wi-Fi is on.
- ❌ Bad: "I remember your Wi-Fi was on earlier, so I'll go ahead with the search."

Stateful Tool Call and Summary

Stateful tool calls—those that modify the environment (e.g., adding calendar events, changing settings)—require **greater caution** than stateless ones.
The assistant should only perform state-changing actions when the user's intent is **explicitly clear**.

If uncertainty exists:

- ✅ Assistant should confirm intent before proceeding.
- ❌ Avoid acting on vague or partial requests.

After the action:

- The assistant must clearly **summarize the result**, including all tool arguments used.

  Example:
  "I added a calendar event titled *Meeting with HR* on *May 23rd at 2 PM* to your *Work* calendar."

Judging Stateful Tool Call Based on Database Changes

When evaluating assistant performance on **stateful tool calls**, it's important to verify that the **tool's output resulted in correct database changes**.

- Ensure arguments passed were valid and complete.
- Ensure no unintended side effects occurred.
- Verify that the state (e.g., calendar, settings) was changed as expected.

This judgment applies **in addition to** evaluating the assistant's natural language response and reasoning.

[round_7_update] Default Clarification Behavior

When the assistant needs to ask the user for missing information, it should:

Only ask for *required arguments*, unless explicitly stated otherwise in the System Message.

❌ **What to Avoid:**

- **Do not** ask the user for technical identifiers or internal parameters like `calendar_id`, `uuid`, etc., especially when users are unlikely to know them.
- Avoid asking for *all* tool parameters by default (especially in tools like `create_calendar_event`).

✅ **What to Do Instead:**

- Surface **only the information a user would reasonably know**, e.g., **calendar name** instead of calendar ID.
- If a required internal value (like `calendar_id`) is needed:
  - Ask for the **friendly name** (e.g., "Which calendar do you want to use?")
  - Then, use another tool (e.g., `search_calendar`) to fetch the corresponding internal value.

This change **differs from our earlier annotation approach**—especially for Calendar tasks—where we previously prompted for multiple parameters (including optional ones). **This should now be avoided.**

Error Labels and Critiques

In the creation process of AC tasks, after we generate the conversation, we also **review the assistant's performance and annotate any mistakes**. This is done via error labels and critique comments, which guide model improvements. While the user (in a real scenario) wouldn't see these, as a conversation designer you need to provide them for any turn where the assistant's response was not ideal initially.

Here are the standard **Error Labels** used to categorize assistant mistakes, with explanations:

- **param_type_inconsistent:** The assistant provided a parameter of the wrong type to a function. For example, a function expects an object or list and the assistant gave a string, or expects a number and got text. *Critique comment example:* "The `calendar_id` should be a string, but the assistant passed an object."

- **param_not_defined:** The assistant used a parameter name that doesn't exist for that tool. *Example:* The assistant tried `gmail_search` with `{ "query": "abc", "folder": "inbox" }` but "folder" is not a defined parameter for that function. *Critique:* "Used an undefined parameter `folder` in gmail_search; that parameter isn't part of the tool definition."

- **extra_param_predicted:** The assistant included an extra parameter or data that wasn't needed or asked for. Slightly different from not defined (the param might exist but contextually wasn't required). Often, this label is used when the assistant guessed some input it shouldn't have. *Example:* The user didn't specify a `place_id`, but the assistant arbitrarily added one in `get_directions`. Or assistant added a search query term that user didn't mention. *Critique:* "The assistant added an unsupported filter `filetype:pdf` in the search query, which wasn't prompted by the user – extra parameter not needed."

- **required_param_missing:** The assistant failed to include a mandatory parameter in a tool call. *Example:* Calling `google_calendar_schedule_event` without a start_time. *Critique:* "Missing `start_time` when scheduling the event – required parameter was omitted."

- **enum_not_respected:** The assistant gave a value that is outside the allowed set for that parameter. Many parameters only accept certain values (enums). *Example:* A function expects `mode` to be one of "walking", "driving", "transit", but the assistant set `"mode": "bicycling"` which might not be allowed (if not in the API's enum). *Critique:* "The `mode` value `bicycling` isn't an allowed option for get_directions (expected walking, driving, or transit).”

- **wrong_tool_selected:** The assistant chose the wrong tool for the job. *Example:* using `google_drive_search` to look up an email. Or using `github_code_search` when the user asked for an issue. *Critique:* “The assistant used the wrong tool – it called `github_code_search` but the user needed information from a pull request (should have used `github_pr_and_issues_search`).”

- **wrong_param_value:** The assistant used an incorrect value for a parameter that *is* of the correct type but logically wrong. *Example:* Searching the wrong keyword (misunderstood query), or using `place_id` of a different place than intended, or selecting wrong index in `open_search_results`. *Critique:* “Wrong value for `open_search_results`: the assistant opened result 3, but based on the conversation it should have opened result 1.”

- **no_tool_triggered:** The assistant should have used a tool in that turn but didn’t. This often happens if the assistant answered from its own knowledge or gave a generic response when actually a tool was needed. *Example:* User: “Do I have any meetings tomorrow?” Assistant: “You have a couple of meetings.” (but it never actually called calendar search – it guessed). That’s a mistake. *Critique:* “The assistant did not call any tool to check the calendar and instead answered directly – it should have used `google_calendar_search` (no tool was triggered).”

- **tool_over_triggered:** The assistant called a tool (or multiple tools) when it wasn’t necessary, or called one too many times. *Example:* User asks a simple question that doesn’t require any tool, but the assistant still calls one. Or the assistant repeats a search it already did with no new info. *Critique:* “Unnecessary tool use: the assistant called `sharepoint_search` twice with the same query even though one call sufficed (tool over-triggered).”

- **parallel_calls_missing:** The situation allowed parallel calls to cover more ground, but the assistant only did one at a time or omitted one path. *Example:* User said search both emails and drive for X; assistant only searched drive in that turn, instead of doing both in parallel or in quick succession. This is a softer issue (not always critical). *Critique:* “The assistant did not use parallel calls for independent searches; it could have searched emails and drive simultaneously for efficiency.”

- **unsatisfactory_summary:** The assistant’s summary of tool results was incomplete, misleading, or incorrect. *Example:* Tool returned a doc with 3 important points, assistant only mentioned 1. Or assistant summary says “the document is about Project X” when it was actually about Project Y (misinterpreted). *Critique:* “The assistant’s summary ignored key details from the document – unsatisfactory summary of the content.”

- **tool_call_not_parsable:** The assistant’s tool call was so badly formatted that the system couldn’t parse/execute it. For instance, missing a quote in JSON, or brackets not balanced. These usually result in errors or no output. *Critique:* “The tool call JSON was malformed and could not be parsed by the system (syntax error).” – In a training scenario, if this happens, we fix the format in the "Fixed Tool Call" and label this error.

- **others:** For any error that doesn’t fit the above categories. If used, always explain clearly. Maybe something like logic mistakes that aren’t covered, or an answer that was irrelevant. *Critique:* “The assistant’s answer was off-topic and didn’t address the user’s request.” (if that happened, though that might also be considered a context issue).

- **no_issues:** This label is used when reviewing a turn to mark that it was perfect (no errors found).

Critic message proposal:

`error_labels` is a list of issues that are present in the model response. They can be from the following taxonomy. Note it is possible to have more than one error label in a particular model response and we expect all applicable labels to be provided.

- `param_type_inconsistent`: e.g. "Argument `target` of function `photos_open_album`` should be a `string` type but tool call gives an `integer` value."
- `param_not_defined`: e.g. "Tool call `photos_open_album` involves the `album` parameter, which is not defined in the tool definition".
- `extra_param_predicted`: Prediction involves an argument that is defined in the tool definition but the ground truth tool call does not specify. e.g. "Should not have used argument `format` for `search_file`."
- `required_param_missing`. Tool definition has an argument that’s required but tool call does not provide it. e.g. "Tool call `photos_open_album` did not provide the `target parameter`, which is a required argument."
- `enum_not_respected`: tool definition specifies a enum type for a parameter but the actual tool call gives a value outside of the enum list. e.g. "The given value `text body` for `paragraph_style` in `notes_set_paragraph_style` is not one of the allowed values."
- `wrong_tool_selected`: e.g. "Tool `photos_open_destination` is not the correct one to use. Should use ``photos_open_album `instead"
- `wrong_param_value`: Parameter errors, hallucination or incompleteness: e.g. "The value of the `notes` parameter in tool `notes_add_tags_to_notes `should be `['CS101', 'CS102', 'TODO']`, not `['CS101']`"
- `no_tool_triggered`: text is given instead of a tool call. e.g. "Answered in text instead of invoking the tool `notes_add_tags_to_notes`"

- `tool_over_triggered`: If a text is expected but got a tool call instead. e.g. "Should ask a follow up question but a tool call to `notes_set_paragraph_style` is triggered"
- `parallel_calls_missing`: If a parallel tool call is expected but a single tool call is predicted. e.g. "Additional functions should have been called in parallel: `get_weather`."
- `unsatisfactory_summary`: If the model tries to summarize the tool call results but makes mistake such as hallucination, misinterpreting the output, or only interpreting one tool output while ignoring others. Describe the issues in `critic_comments`.
- `tool_call_not_parsable`: If the model fails to generate a parsable tool call. `critic_comments` can be empty in this case.
- `others`: other errors. Describe in `critic_comments` .
- `no_issues`: no issues. `critic_comments` can be empty in this case.

`critic_comments` is in free text natural language. It should be specific about the issues. e.g. Instead of saying "Wrong tool is chosen", it should say "Tool `photos_open_destination` is not the correct to use. Should use `photos_open_album` instead".

When you identify an error in a turn:

1. **Mark the error label(s)** for that turn. There can be more than one label if, say, multiple things went wrong. For example, the assistant might have chosen the wrong tool *and* left out a required parameter – you'd mark both `wrong_tool_selected` and `required_param_missing`.

2. **Write a critic comment** (in a separate "critic" section or interface field) describing what was wrong and what the assistant should have done. This should be in natural language, concise but specific. Use a neutral or instructive tone, not scolding. Essentially, you're writing feedback to the model:

    - "It used `search_place` and got a place_id, but never used `get_directions` even though the user asked for directions. It should have followed up with `get_directions` (no_tool_triggered)."
    - If multiple errors, you can list them or separate by newline. The interface might allow each error label to have its own comment.

After labeling and commenting, you then provide the **fixed turn**:

- **Fixed Assistant Response:** If the original assistant message was wrong or incomplete (content-wise or tone-wise), you rewrite it here the way it *should* have been. This is essentially what we include in the final polished conversation. For instance, removing a phrase, adding missing info, correcting a statement.
- **Fixed Tool Call:** If the issue was with the function call, you put the corrected function call here. E.g., correct the parameters, or replace the wrong function with the right one. If the fix is that no tool should have been called, then the "fixed tool call" might actually be empty (meaning the assistant shouldn't have made a call at that point). Or vice versa, if the error was not calling a tool, the fixed tool call field would now include the call that should have been made.

Example:
User: "Schedule a meeting with Bob tomorrow at 3 PM."
Assistant (original, errored): *Calls* `google_calendar_schedule_event` *but forgets to include Bob in attendees or doesn't specify which calendar.*

This has errors: missing required param (attendees) and maybe assumed a calendar.

- Error labels: `required_param_missing`, extra_param_predicted.
- Critic comment: "The assistant scheduled the event without adding Bob as an attendee (missing required attendee parameter) and didn't confirm which calendar to use (assumed default calendar without asking). It should ask which calendar and include Bob's email in attendees."
- Fixed Assistant Response: "Certainly. Do you want this meeting on your Work or Personal calendar? I'll include Bob as an attendee and set it for tomorrow at 3 PM once you confirm the calendar."
  (We choose to fix by asking which calendar, as that's the safer solution.)
- Fixed Tool Call: *Would be empty here in the first turn fix, because we decided the assistant should ask first rather than calling immediately. If we instead assumed a calendar but included Bob properly, the fixed call might be the corrected JSON with Bob's email.*

This level of detail in critique ensures that when the data is used to fine-tune the model, it learns from these mistakes.

For our final playbook purposes, remember these error labels and avoid having any of these issues in the conversations you deliver. We include them here so you understand what *not* to do.

To ensure quality and traceability of assistant behavior during annotation, it's essential that you understand and properly use the following fields when reviewing assistant responses:

---

1. `error_labels` – What went wrong?

- This is a **list of predefined error types** that describe what kind of mistake the model made in a tool call or assistant message.
- **Multiple error labels** can be applied to a single assistant response if needed.

Common Examples:

- `param_type_inconsistent`: wrong data type used
- `required_param_missing`: a required field was skipped
- `wrong_tool_selected`: the tool used was not appropriate for the request
- `unsatisfactory_summary`: tool result was misinterpreted or hallucinated
- `no_issues`: no error—model performed as expected

---

2. `critic_comments` – Why was it wrong?

- This is a **freeform explanation** written by the trainer in **3rd person** style.
- It explains **exactly what the error was and why it matters**.
- Should be **specific**, **concrete**, and **match the order** of the `error_labels`.

Avoid vague comments like:

- "Wrong tool"
- "Param missing"

✅ Instead, be **precise** so the issue can be clearly understood and fixed.

---

3. `reasoning_comment` – Why did the assistant do this?

- This is added **in human corrected responses only**.
- It represents the **assistant's internal logic**, like a **"thought bubble"**.
- Written in **first person**, as if the assistant is **explaining its own reasoning**.

Good Example:

> "Since West is currently pointing to the top of the image, I need to rotate the photo clockwise so South points up, as the user requested."

Bad Example:

> "The assistant rotated the image clockwise." ← *(Third person – incorrect)*

Compliance

By now, many guidelines have been covered. This is a quick **summary checklist** of the most critical do's and don'ts that calibrators especially look for:

- ✅ **DO make user prompts sound human and scenario-based.** They should reflect real needs someone might have, given the toolset. For example, a user in a corporate scenario might ask about "the latest audit report on SharePoint" or "schedule a client call next week," whereas a personal user might ask to "find my flight confirmation email." Tie the prompt to a realistic persona and context, and write it as that persona would actually speak or text.
- ✅ **DO push the assistant to perform multi-step reasoning.** If a task can be solved with just one step but could be made more instructive with a second step, add complexity. This demonstrates richer interaction.
- ✅ **DO maintain logical progression.** Each new user utterance should either (a) refine a previous request, (b) ask a follow-up based on the assistant's last answer, or (c) start a new but related sub-task. Avoid non-sequiturs unless intentionally introducing a new scenario (which is generally not done mid-conversation).
- ✅ **DO clarify rather than assume.** When in doubt, ask the user for more info. It's much better to have a turn where the assistant says "Can you clarify X?" than to have a turn where the assistant guessed wrong.
- ✅ **DO use bullet points or numbering in assistant answers where it improves clarity.** Particularly if listing multiple items or steps. It's easier to read and is explicitly encouraged.
- ✅ **DO test the conversation mentally:** Imagine yourself as the user reading it. Does the user prompt sound like something you'd say? Does the assistant's reply make sense and is it helpful? Did the assistant answer the question fully? If you spot anything that feels off, fix it before submitting.

- ❌ **DO NOT use overly simplistic prompts or questions that don't utilize tools.** For example, a user question like "How many days are in a year?" – these are trivial and don't exercise the tool usage (and if it's a general chat, it's too trivial to be interesting). If you include general chat tasks, make them at least somewhat linked to the domain or requiring a bit of thought (e.g., "What local events can I attend in the Bay Area this weekend?" – the assistant could use calendar tool to see events or just general knowledge, it's borderline, but at least it's richer than a basic fact).
- ❌ **DO NOT let the assistant output contradict previous info.** For example, if earlier it said the user's name is Alice, later it shouldn't refer to the user as Bob. Or if earlier a tool result showed something, the assistant shouldn't later say something that conflicts with that. Keep track of details to ensure consistency.

Strict Fail Conditions

Certain mistakes are considered so severe that a single instance can cause the whole task to be rejected immediately. Avoid these at all costs:

- **Using an unapproved tool or API:** As said, if you call anything outside the allowed list (or even mention doing so), it's a fail.
- **No tools used in a feasible/infeasible task:** If you label a task as Feasible but then produce a conversation where the assistant never calls a tool (and just chats or answers from knowledge), you've fundamentally failed the task purpose.
- **Very unnatural user prompt or interaction:** If the user's first prompt is clearly violating the naturalness guidelines (like the earlier example stuffing in technical terms or being extremely long and formal), the task can be rejected. The user side is within our control, so there's little excuse for it to be wrong.
- **Assistant output that is nonsensical or irrelevant:** If at any point the assistant's response doesn't make sense (due to a mistake in designing it), that's a serious issue. E.g., assistant answers a different question than asked, or babbles.
- **Major factual error or hallucination:** Especially if the user asks for something from the tools and the assistant claims to have found it but in reality that info isn't in the tool results – that's a big no-no. Essentially lying about the content of the data.
- **System Message** should be unique and complex enough.

Common Errors and Possible Solutions:

1. Error 400: Service Temporarily Unavailable
   1.1. Check whether all the previous cells are either accepted or rejected, if not then do it.
   1.2. Do a hard refresh and then check the same thing again.
   1.3. Try deleting and recreating assistance response, and wait each time after a cell is accepted for it to be saved completely. likewise accept each cell one by one.
   1.4. Sometimes

```
{
  "success": true
}
```

   gets added in tool output as something extra, so remove it from Colab, and it should work fine.
   1.5. Check if you are using any Reddit-related tool in between your conversation; if yes, then remove it or use it in the end since its output is too long and hence we exceed the token limits.
   1.6. Do not call any tool in parallel with state-related tools.
      - get_system_settings
      - set_system_settings

- get_current_location
- get_cellular_status
- get_wifi_status
- get_location_service_status
- get_low_battery_mode_status
- get_locale
- get_timezone_utc_offset_seconds
- get_current_latitude
- get_current_longitude
- get_current_place_id
- get_current_formatted_address
- set_cellular_status
- set_wifi_status
- set_location_service_status
- set_low_battery_mode_status
- set_locale
- set_timezone_utc_offset_seconds
- get_system_settings_field
- get_current_location_field

2.

Final Notes

This playbook has integrated the latest guidelines as of 2025. By following this playbook, you should be able to create Agent Completion conversations that are **detailed, realistic, and high-quality**, meeting the client's requirements. Use it as a reference while working on tasks, and don't hesitate to refer back to it for any doubt about how to handle a certain situation. Good luck, and happy conversing!