

Use Case 2: Meeting Prep Agent

- **Overview:**

As a Grabber, I want to have an intelligent agent automatically gather and summarise all relevant information before my upcoming meetings—by analysing the calendar invite, attached files, past meeting notes, and related conversations—and present it to me in a single, digestible brief, so that I can walk into any meeting fully prepared and ready to contribute effectively, without spending a lot of time manually searching for context.

Expanded Description :

This agent's value comes from its ability to synthesise information from multiple sources:

- **Calendar Context:** Parses the Google Calendar invitation for the agenda, attendee list, and description.
- **Attached Documents:** Identifies and provides direct access to any files attached to the invite (e.g., PDFs, Google Docs).
- **Historical Context:** References past occurrences of recurring meetings to find previous notes, documents, or Zoom transcripts to surface decisions and open action items.
- **Conversational Context:** Scans relevant Slack channels for recent discussions related to the meeting's topic or attendees.
- **Broad Search:** Conducts internal searches (e.g., on Confluence or Google Drive) for related project documents and external searches for information on external attendees or their companies.
- **Brief:** Delivers this synthesised brief in a dedicated "AgentSpace" window for easy access.

Acceptance Criteria (AC)

GIVEN my 4:30 PM meeting is a recurring "Project Phoenix Weekly Sync," WHEN the agent prepares the brief, THEN it must also include a link to last week's meeting notes/transcript and a summary from previous sessions.

GIVEN I have a meeting in my Google Calendar at 4:30 PM with an agenda and **an attached presentation or any other documents**, WHEN the Meeting Prep Agent activates (e.g., 30 minutes before the meeting), THEN the AgentSpace window must display a summary of the agenda and a **direct link to the documents**.

GIVEN the meeting title contains "Project Phoenix," and there is a #project-phoenix Slack channel, WHEN the agent prepares the brief, THEN it must include a summary of the most recent, relevant messages from that channel, in relation to the meeting title.

GIVEN I have a meeting scheduled, WHEN the agent runs but cannot find any relevant attachments, previous meetings, or discussions, THEN it must inform me in the AgentSpace that "No preparatory materials could be found for this meeting."

GIVEN a brief has been prepared for my meeting, WHEN the agent adds the related documents to the AgentSpace window, THEN each document must be clearly labeled and link directly to the source file.

- **Feasibility:** High. This is a high-value use case for agentic systems. The required integrations are common and the value to the user is immediate and obvious.
- **Potential Solution:**
 - **ADK and AgentSpace:** An agent can be triggered by an upcoming calendar event. Agent needs to check the calendar but can not see the attachments, links. Workaround: save all attachments to drive and add drive to datastore.
 - **Connectors:** The agent would connect to:
 - Google Calendar to get meeting details, attendees, and attachments.
<https://cloud.google.com/agentspace/agentspace-enterprise/docs/connect-calendar>
 - Google Drive and/or other document repositories to access meeting-related files.
<https://cloud.google.com/agentspace/agentspace-enterprise/docs/connect-google-drive>
 - Slack or other chat tools to search for relevant conversations.
<https://cloud.google.com/agentspace/agentspace-enterprise/docs/connect-slack>
 - **Logic:** The agent would use the meeting title, attendees, and agenda as search queries across the connected data sources. The LLM would then summarize the findings into a briefing document.
- **? Clarification Questions for the Customer Kickoff Meeting**

User Experience & Scope

1. **Trigger & Timing:** The document suggests a 30-minute pre-meeting trigger. Is this the ideal timing for all users? Should this be a configurable setting for each user? Would users also want a button to manually generate the brief on-demand?
2. How far back in time should the agent search for relevant information?
3. **Briefing Format:** The goal is a "digestible brief". Can you describe the ideal layout in AgentSpace? For example, should it be a single narrative summary, or organized into tabs like "Agenda," "Key Documents," "Recent Slack Chat," and "Action Items"? Is it briefing docs showing on AgentSpace?
4. **Historical Search Depth:** When searching for past meetings or conversations, how far back should the agent look? Should it be limited to the last 7 days, the time since the last recurring meeting, or a different time frame?

Data Sources & Integrations

5. **Tooling Confirmation:** The document mentions Google Calendar, Google Drive, and Slack. Are there any other data sources we need to consider, such as Confluence, Jira, or Microsoft Teams?
6. **Calendar Attachment Workaround:** The current technical limitation requires saving calendar attachments to Google Drive for the agent to access them. Is this manual step of saving attachments to Drive an acceptable workflow for your users?
7. **"Relevant" Conversations:** The agent needs to find "relevant messages" in Slack. How do we define "relevance"? Is matching keywords from the meeting title enough, or do we need a more advanced understanding of the conversation's context?
8. **External Attendees:** The agent is expected to perform "external searches" for information on attendees. What are the approved sources for this (e.g., LinkedIn, corporate websites, news articles)? We need to be mindful of privacy and the reliability of information.

Security & Privacy

9. **Permissions Model:** How should the agent handle permissions? Will it operate using the individual user's credentials to respect their access rights, or will it use a service account? This is critical for accessing private documents and Slack channels.
10. **Data Handling:** What are the data retention policies for the generated briefs? Should the brief be ephemeral and disappear after the meeting, or should it be stored for future reference?
11. Are there any privacy or access restrictions for searching chat channels or documents?

Product Requirements Document (PRD): Meeting Prep Agent

This PRD translates the customer's use case into a structured document for your development team.

1. Overview

The Meeting Prep Agent is an intelligent assistant designed to automatically gather, synthesize, and summarize all relevant information for a user's upcoming meetings. By analyzing calendar invites, documents, historical context, and conversations, the agent delivers a concise brief in Google AgentSpace. The primary goal is to empower users to walk into any meeting fully prepared and ready to contribute effectively, drastically reducing manual preparation time.

2. User Persona

- **The User ("A Grabber"):** A busy professional who attends multiple meetings daily. They need to quickly get up to speed on meeting topics, review past decisions, and understand the context of discussions without spending significant time searching across different applications.

3. Functional Requirements (Features)

Feature ID	Feature Name	Description	Acceptance Criteria
F-01	Calendar Integration	The agent will connect to the user's Google Calendar and parse upcoming events.	- Must extract meeting title, time, attendees, and description/agenda. - Must be triggered automatically a set time before the meeting (e.g., 30 mins).

F-02	Document Retrieval	The agent will identify and provide links to all documents relevant to the meeting.	<ul style="list-style-type: none"> - Must link to any documents attached to the invite (via the Drive workaround).
 - Must search Google Drive for related project documents based on the meeting title.
 - Each document in the brief must be clearly labeled and link to the source file. 	
F-03	Historical Context	For recurring meetings, the agent will surface information from past sessions.	<ul style="list-style-type: none"> - Must find and link to the previous meeting's notes, documents, or transcript.
 - Must summarize key decisions and open action items from past sessions. 	
F-04	Conversational Context	The agent will scan Slack for recent, relevant discussions.	<ul style="list-style-type: none"> - If a meeting title is "Project Phoenix," the agent must scan the 	#project-phoenix Slack channel. - Must include a summary of the most recent and relevant messages related to the meeting topic or attendees.
F-05	Brief Synthesis & Delivery	The agent will use an LLM to synthesize all gathered information into a single brief	<ul style="list-style-type: none"> - The brief must be delivered in a dedicated AgentSpace window.
 - If no materials are found, the agent must clearly state: "No preparatory materials could be found for this meeting". 	

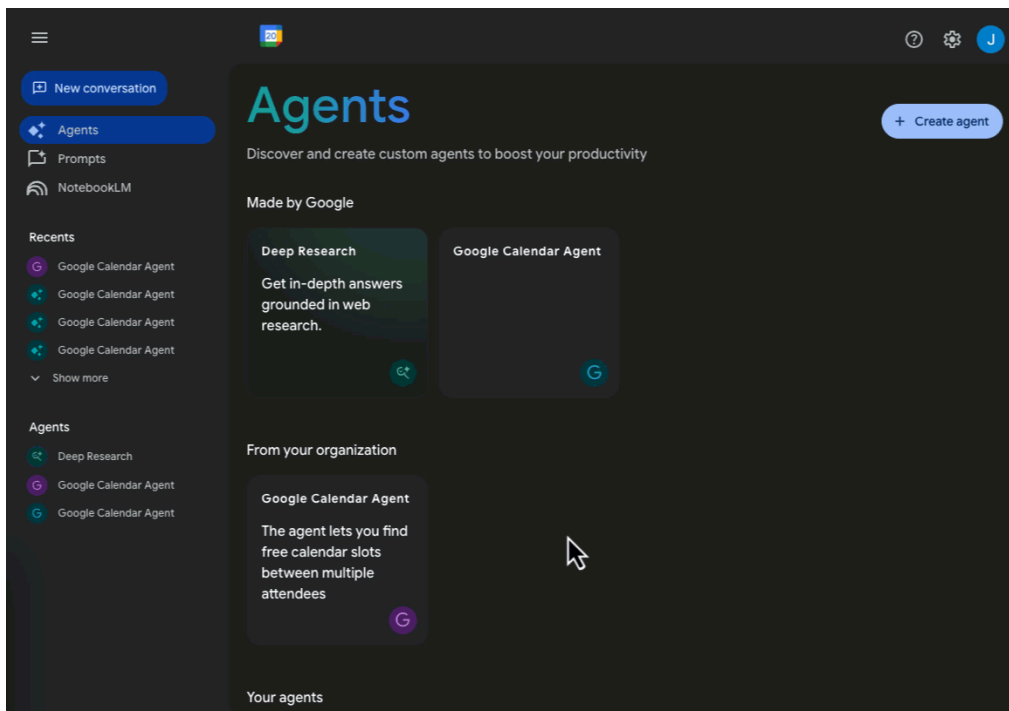
and present it in AgentSpace.

4. Technical Requirements

- **Platform:** The solution will be built using the Google Agent Development Kit (ADK).
- **Deployment:** The agent will be deployed on Google Cloud Agent Engine.
- **User Interface:** The final briefing will be displayed within Google AgentSpace.
- **Connectors:** The agent will require connectors to:
 - Google Calendar
 - Google Drive
 - Slack

5. Assumptions and Dependencies

- Users will grant the agent the necessary OAuth permissions to access their Calendar, Drive, and Slack data.



Purpose:

This document provides a guide to leveraging AgentSpace's built-in OAuth handling for authenticating an ADK-based agent deployed on Agent Engine. It outlines the process of

obtaining and reusing an access token within the agent. This token then enables the agent to securely interact with a range of Google services, including Google Calendar, Google Drive, User Profile endpoints, and other GCP-scoped resources, allowing it to perform actions on behalf of the user.

Prerequisites:

Before proceeding, you must ensure the following are set up:

- Create an OAuth 2.0 Authorization: You will need to create an OAuth 2.0 authorization within your Google Cloud project. This involves defining the `CLIENT_ID`, `CLIENT_SECRET`, and `SCOPES` required for your agent to access Google services. The `create_authorization.sh` script provides a template for this process.

```
#!/bin/bash
```

```
# --- Variables ---
```

```
PROJECT_ID="" # Your Google Cloud Project ID
```

```
AUTHORIZATION_ID="" # A unique ID for the authorization, e.g., "gcal-agent-auth"
```

```
CLIENT_ID="" # The OAuth 2.0 Client ID from your Google Cloud project
```

```
CLIENT_SECRET="" # The OAuth 2.0 Client Secret from your Google Cloud project
```

```
SCOPES="" # A space-separated list of OAuth scopes, e.g.,  
"https://www.googleapis.com/auth/calendar.readonly  
https://www.googleapis.com/auth/userinfo.email"
```

```
DISCOVERY_ENGINE_API_BASE_URL="https://discoveryengine.googleapis.com/v1alpha"
```

```
# --- Script Body ---
```

```
AUTH_TOKEN=$(gcloud auth print-access-token)
```

```
# Construct the authorizationUri separately and URL-encode the scopes
```

```
# Using printf %s to avoid issues with newline in variable expansion
```

```
ENCODED_SCOPES=$(printf %s "${SCOPES}" | sed 's/ /+/g') # More robust way to replace  
spaces with +
```

```
AUTHORIZATION_URI="https://accounts.google.com/o/oauth2/v2/auth?&scope=${ENCODED_SCOPES}&include_granted_scopes=true&response_type=code&access_type=offline&prompt=consent"
```

```
# Create the JSON payload
```

```
JSON_PAYLOAD=$(cat <<EOF
```

```
{
```

```
  "name": "projects/${PROJECT_ID}/locations/global/authorizations/${AUTHORIZATION_ID}",
```

```
  "serverSideOauth2": {
```

```
    "clientId": "${CLIENT_ID}",
```

```
    "clientSecret": "${CLIENT_SECRET}",
```

```
    "authorizationUri": "${AUTHORIZATION_URI}",
```

```
    "tokenUri": "https://oauth2.googleapis.com/token"
```

```
  }
```

```
}
```

```
EOF
```

```
)
```

```
curl -X POST \
```

```
  -H "Authorization: Bearer ${AUTH_TOKEN}" \
```

```
  -H "Content-Type: application/json" \
```

```
  -H "X-Goog-User-Project: ${PROJECT_ID}" \
```

```
  "${DISCOVERY_ENGINE_API_BASE_URL}/projects/${PROJECT_ID}/locations/global/authorizations?authorizationId=${AUTHORIZATION_ID}" \
```

```
  -d "${JSON_PAYLOAD}"
```


- Create an Agent: After setting up the authorization, you must create an agent that utilizes this authorization. The `create_agent.sh` script demonstrates how to create an agent, linking it to the previously defined authorization using the `AUTH_ID`. This agent will be the entity that performs actions on behalf of the user.

```
#!/bin/bash
```

```
# --- Variables ---
```

```
PROJECT_ID="" # Your Google Cloud Project ID
```

```
PROJECT_NUMBER="" # Your Google Cloud Project Number
```

```
LOCATION="" # The location of your resources, e.g., "us-central1"
```

```
AS_APP="" # The application identifier, e.g., "gcal-agent_1747246671415"
```

```
ASSISTANT_ID="" # The assistant ID, e.g., "default_assistant"
```

```
AGENT_NAME="" # A unique name for the agent, e.g., "gcal-agent-2"
```

```
AGENT_DISPLAY_NAME="" # The display name for the agent, e.g., "Google Calendar Agent"
```

```
AGENT_DESCRIPTION="" # A brief description of what the agent does
```

```
TOOL_DESCRIPTION="" # A description of the tool used by the agent
```

```
REASONING_ENGINE="" # The full resource name of the reasoning engine, e.g.,  
"projects/<PROJECT_NUMBER>/locations/<LOCATION>/reasoningEngines/<REASONING_ENGI  
NE_ID>"
```

```
AUTH_ID="" # The authorization ID, e.g., "gcal-agent-auth"
```

```
DISCOVERY_ENGINE_API_BASE_URL="https://discoveryengine.googleapis.com"
```

```
# --- Script Body ---
```

```
AUTH_TOKEN=$(gcloud auth print-access-token)
```

```
curl -X POST \
```

```
-H "Authorization: Bearer ${AUTH_TOKEN}" \
```

```
-H "Content-Type: application/json" \
```

```
-H "X-Goog-User-Project: ${PROJECT_ID}" \
```

```
"${DISCOVERY_ENGINE_API_BASE_URL}/v1alpha/projects/${PROJECT_ID}/locations/global/collections/default_collection/engines/${AS_APP}/assistants/${ASSISTANT_ID}/agents" \
```

```
-d '{
```

```
  "name":
```

```
  "projects/"${PROJECT_NUMBER}"/locations/"${LOCATION}"/collections/default_collection/engines/"${AS_APP}"/assistants/"${ASSISTANT_ID}"/agents/"${AGENT_NAME}",
```

```
  "displayName": ""${AGENT_DISPLAY_NAME}""
```

```
  "description": ""${AGENT_DESCRIPTION}""
```

```
  "adk_agent_definition": {
```

```
    "tool_settings": {
```

```
      "tool_description": ""${TOOL_DESCRIPTION}""
```

```
    },
```

```
    "provisioned_reasoning_engine": {
```

```
      "reasoning_engine": ""${REASONING_ENGINE}""
```

```
    },
```

```
    "authorizations": [
```


```
      "projects/"${PROJECT_NUMBER}"/locations/global/authorizations/"${AUTH_ID}""
```

```
    ]
```

```
  }
```

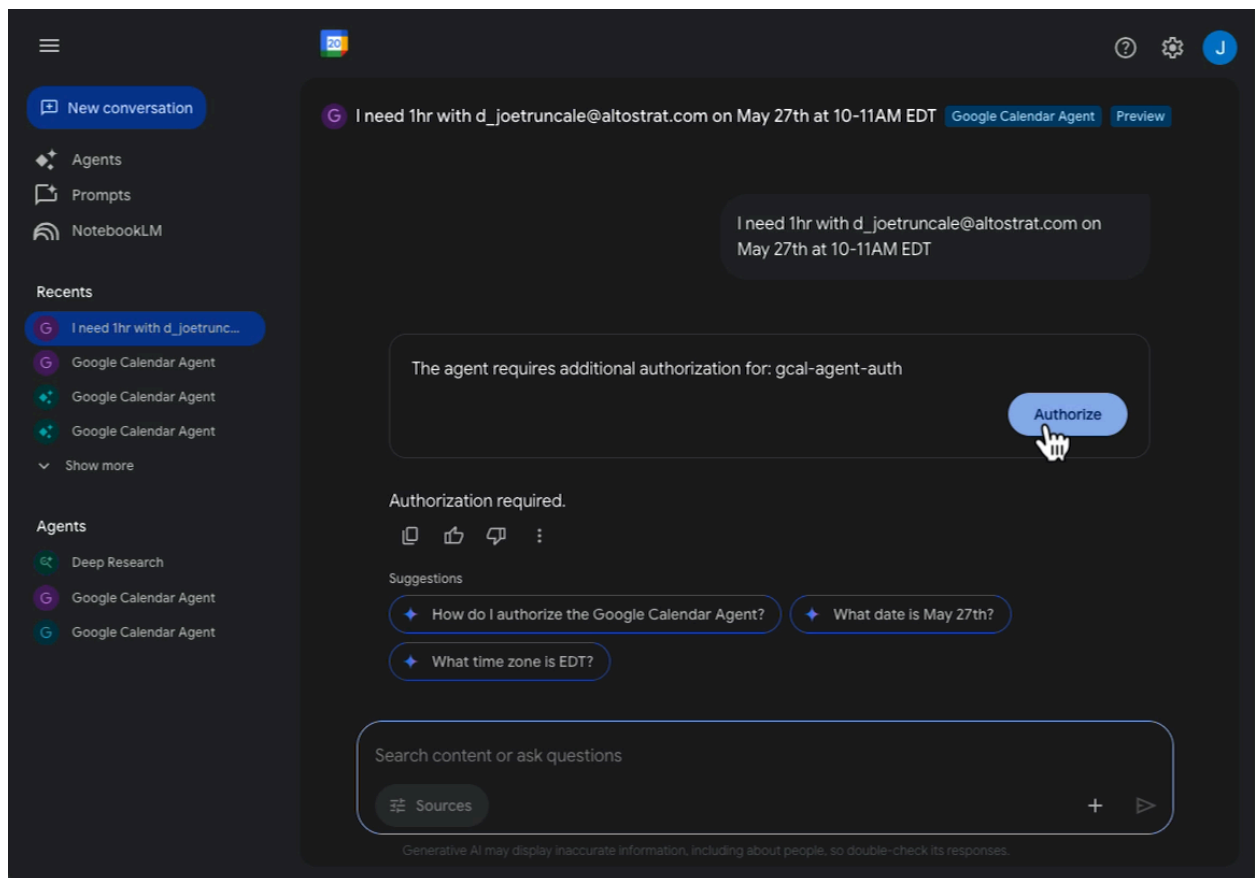
```
}'
```

Scripts can be found here: [GitLab](#) or [Google Drive](#)

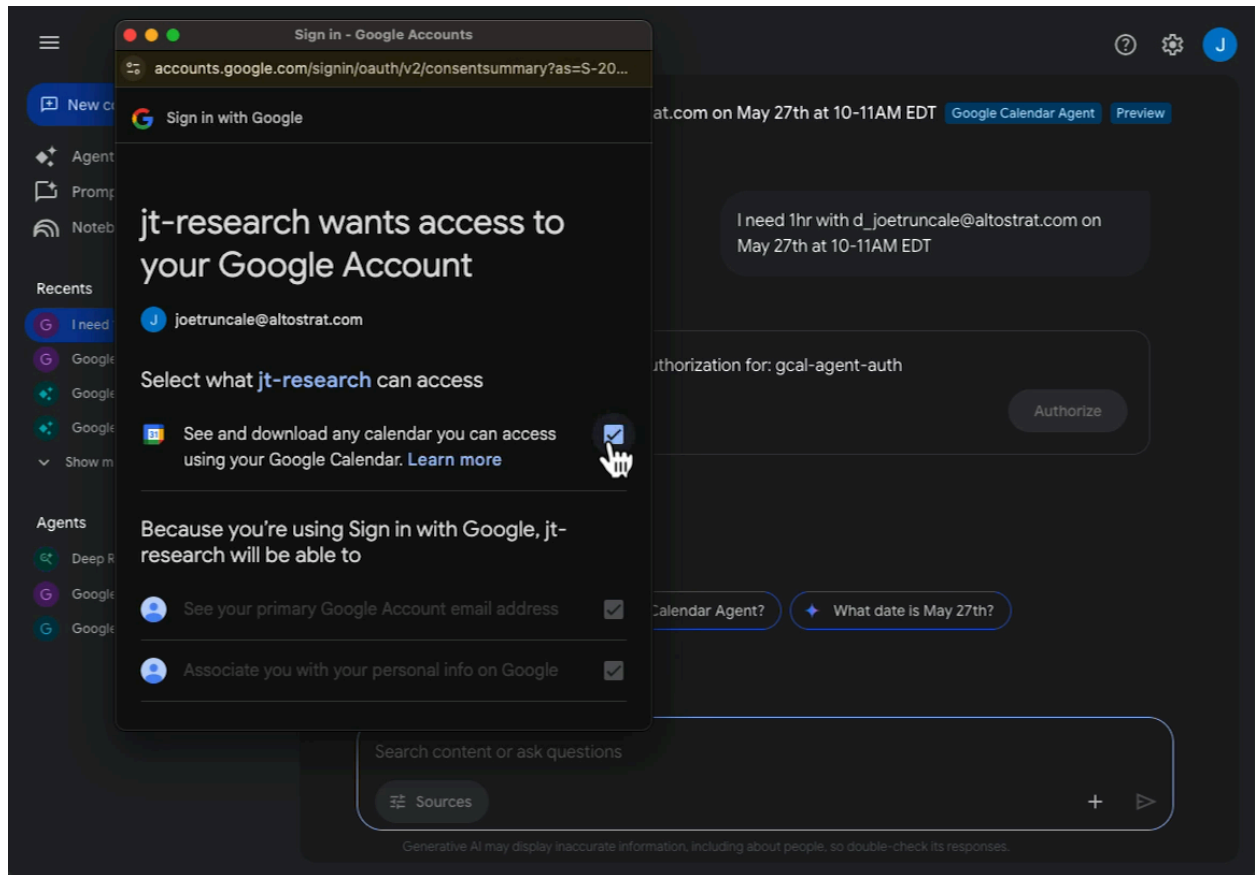
More detailed information:  [How to register and use ADK Agents with Agentspace.pdf](#)

Guide:

Once you create/register your agent with authorization defined, you'll see this oAuth flow as soon as you trigger the agent that you've created.



Make sure you select all of the scopes that are needed from the oAuth consent screen.

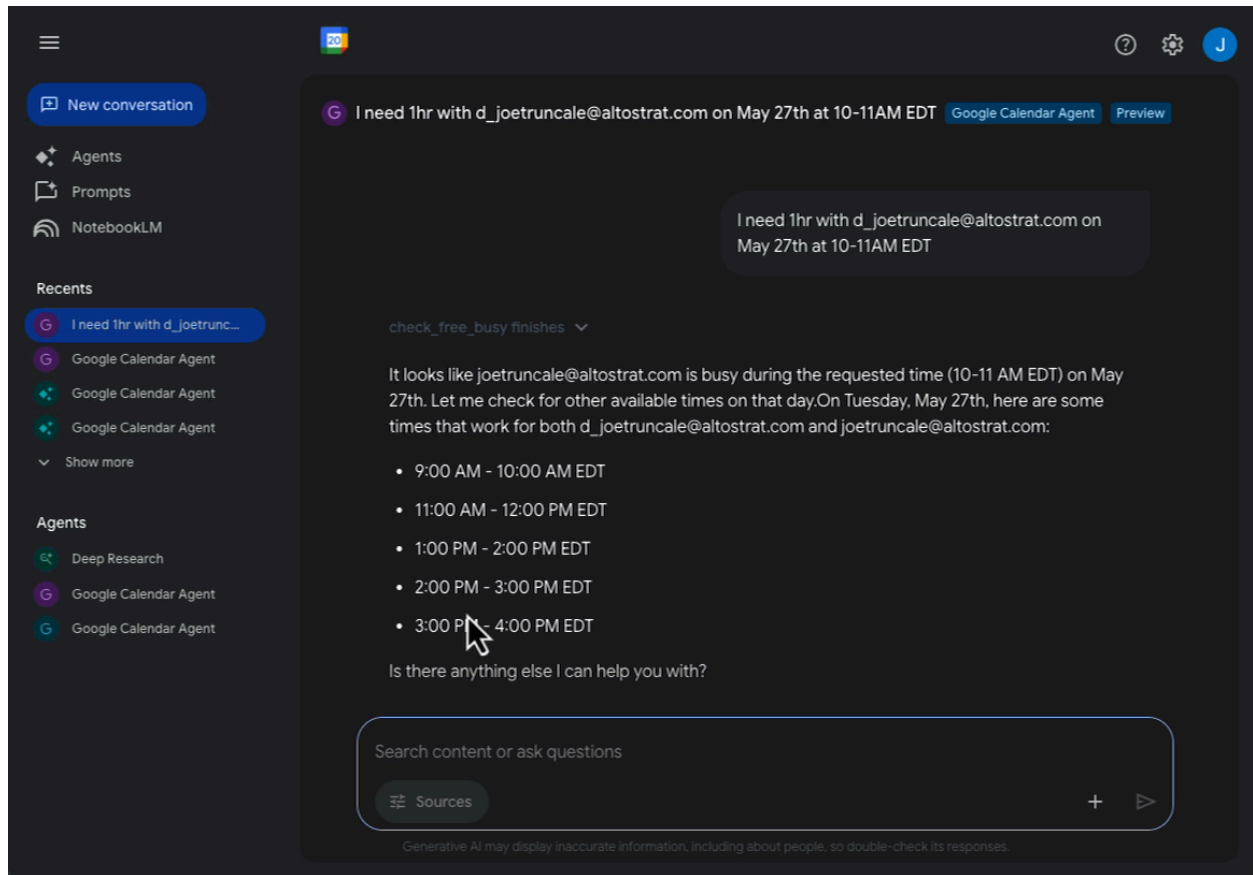


Once the OAuth handshake is complete, the access token is managed by Agentspace in terms of expiration and refreshing (source: [HTTP Agents for Agentspace](#)). That token is [stored in the context](#) for your agent (ToolContext, CallbackContext, etc..). There's a `temp` prefix for these tokens so that there isn't an accidental collision in terms of keys.

Python

```
auth_id = '' # the authorization you created in your prerequisite
access_token = tool_context.state[f"temp:{auth_id}"]
creds = Credentials(token=access_token)
service = build('calendar', 'v3', credentials=creds)
```

Now, whenever you need to call a GCP service that requires credentials, you can pull that access token from whichever context you're currently using.



If you'd like to try out the functionality, here's the [Google Calendar Agent](#) (CE GitLab) that was built to support this. If you don't have access, [here's a zip in Google Drive](#).

Sample Code for Agent:

```
import os.path
from datetime import datetime
from typing import Dict
from typing import List
from dotenv import load_dotenv

import vertexai
from vertexai.preview import reasoning_engines
from vertexai import agent_engines

from google.adk.agents import LlmAgent
from google.adk.tools.tool_context import ToolContext
from google.adk.agents.callback_context import CallbackContext
from google.oauth2.credentials import Credentials
from googleapiclient.discovery import build

# Load environment variables from .env file
```

```
load_dotenv()
```

```
print("loading .env")
```

```
google_cloud_project = os.getenv("GOOGLE_CLOUD_PROJECT")
google_cloud_location = os.getenv("GOOGLE_CLOUD_LOCATION")
staging_bucket = os.getenv("STAGING_BUCKET")
auth_id = os.getenv("AUTH_ID")
agent_display_name = os.getenv("AGENT_DISPLAY_NAME")
```

```
def current_datetime(callback_context: CallbackContext):
    # get current date time
    now = datetime.now()
    formatted_time = now.strftime("%Y-%m-%d %H:%M:%S")
    callback_context.state["_time"] = formatted_time
```

```
def whoami(callback_context: CallbackContext, creds):
    user_info_service = build('oauth2', 'v2', credentials=creds)
    user_info = user_info_service.userinfo().get().execute()
    user_email = user_info.get('email')
    callback_context.state['_user_email'] = user_email
```

```
calendar_service = build('calendar', 'v3', credentials=creds)
# Get the user's primary calendar to find their timezone
calendar_list_entry = calendar_service.calendarList().get(
    calendarId='primary').execute()
user_timezone = calendar_list_entry.get('timeZone')
callback_context.state['_user_tz'] = user_timezone
```

```
print(f"User's primary calendar timezone: {user_timezone}")
```

```
def prereq_setup(callback_context: CallbackContext):
    print("**** PREREQ SETUP ****")
    access_token = callback_context.state[f"temp:{auth_id}"]
    creds = Credentials(token=access_token)
    current_datetime(callback_context)
    whoami(callback_context, creds)
```

```
def check_free_busy(attendee_emails: List[str],
                    start_time: str,
```

```

        end_time: str,
        tool_context: ToolContext,
    ):

print("check_free_busy in progress...")

try:
    formatted_attendee_emails: List[Dict[str, str]] = []
    if not isinstance(attendee_emails, list):
        raise TypeError("Input must be a list.")
    for email in attendee_emails:
        if not isinstance(email, str):
            raise TypeError("Each element in the input list must be a string.")
        formatted_attendee_emails.append({"id": email})
        print("Email: " + email)

    service = None

    try:
        # Attempt to build the service
        access_token = tool_context.state[f"temp:{auth_id}"]
        creds = Credentials(token=access_token)
        service = build('calendar', 'v3', credentials=creds)

        print("Google Calendar API service built successfully.")

    except Exception as e:
        # Catch any other unexpected errors during the build process
        print(f"An unexpected error occurred: {e}")

    # service = build('calendar', 'v3')
    free_busy_request_body = {
        'timeMin': start_time,
        'timeMax': end_time,
        'items': formatted_attendee_emails
    }
    print(f"Request Body: {free_busy_request_body}")

    freebusy_result = service.freebusy().query(
        body=free_busy_request_body).execute()

    print(f"Response Body: {freebusy_result}")
    return freebusy_result

```

```
except Exception as e:
    print(f'An error occurred: {e}')
    return None
```

```
check_availability = LlmAgent(
    name="check_availability",
    model="gemini-2.5-flash-preview-04-17",
    description="Handles calendar scheduling",
    # flow="auto",
    instruction="""
```

You are specialized in google calendar scheduling to find available timeslots for all attendees.

Assumptions:

- Their working hours are 9AM-5PM in their timezone on weekdays (no weekends) and they only want to schedule during those times unless explicitly told otherwise.

Do not check weekends for availability. You accept both single dates and date ranges.

- You can use date ranges and relative dates, no need for exact dates

Follow these guidelines:

- Do not greet user, Do not welcome user as you already did.

1. Check calendar availability

- Gather User Information:

- Check if you have all of the attendeeEmails including from the conversation context.
- If not, politely ask for their attendeeEmails.
- The timezone is {_user_tz}
- The current datetime is {_time}
- My email is {_user_email} and you should add it to attendeeEmails by default
- Check if you have the date and time for this meeting in the conversation context
- If not, politely ask them for a date or time range.
- Whatever format they provide, use ISO8601 format internally
- Confirm if they want to proceed with scheduling a meeting. If yes, move to the next

step.

- Check Attendee Availability

- If you have the attendeeEmails use the check_free_busy tool to get busy timeslots.

Make sure date and time are ISO8601

- Once you have the busy timeslots, identify the free timeslots. We are looking for overlapping free timeslots that apply to ALL attendees✓✓✓

- If there are no overlapping free timeslots for all attendees during the provided timeframe, tell the user and suggest alternative times using the same dates when they are all free

- Inform User:

- Present up to 5 overlapping free timeslots as a bulleted list formatted in their timezone.

- Ask "Is there anything else I can help you with?"

If you are unable to assist the user or none of your tools are suitable for their request, transfer to other child agents.

```
"""
tools=[check_free_busy],
before_agent_callback=prereq_setup,
)

root_agent = LlmAgent(
    model="gemini-2.0-flash",
    name="root_agent",
    instruction="""
        You are a helpful virtual assistant for Google to help users by
        checking and suggesting Google Calendar availability. If user asks you
        to check calendar availability, please always use "check_availability"
        to check and propose an available timeslots. Never greet user
        again if you already did previously."""
    sub_agents=[check_availability]
)

def deploy_agent_engine_app():
    app = reasoning_engines.AdkApp(
        agent=root_agent,
        enable_tracing=True,
    )

    vertexai.init(
        project=google_cloud_project, # Your project ID.
        location=google_cloud_location, # Your cloud region.
        staging_bucket=staging_bucket, # Your staging bucket.
    )

    agent_config = {
        "agent_engine": app,
        "display_name": agent_display_name,
        "requirements": "requirements.txt",
        # "extra_packages": [".env"]
    }

    existing_agents = list(
        agent_engines.list(filter=f'display_name="{agent_display_name}"'))
```

```

if existing_agents:
    print(f"Number of existing agents found for {agent_display_name}:" + str(
        len(list(existing_agents))))
    print(existing_agents[0].resource_name)

```

```

if existing_agents:
    # update the existing agent
    remote_app = agent_engines.update(
        resource_name=existing_agents[0].resource_name, **agent_config)
else:
    # create a new agent
    remote_app = agent_engines.create(**agent_config)

```

```

return None

```

```

if __name__ == "__main__":
    deploy_agent_engine_app()

```

- The workaround of users saving calendar attachments to a specified Google Drive folder is an acceptable interim solution.
- The customer's primary document storage is Google Drive and their primary chat tool is Slack, as specified in the document.

6. Out of Scope

- The agent will **not** modify or create any events on the user's calendar.
- The agent will **not** send messages or interact with other users on behalf of the primary user.
- This initial version will **not** support data sources beyond Google Workspace and Slack unless specified after the kickoff.

7. Success Metrics

- **User Adoption:** Percentage of target users who actively use the agent for their meetings.
- **Time Saved:** Reduction in self-reported time spent preparing for meetings.
- **Task Success Rate:** Percentage of meetings for which a brief is successfully generated and delivered on time.
- **User Satisfaction:** Qualitative feedback and satisfaction scores from users.