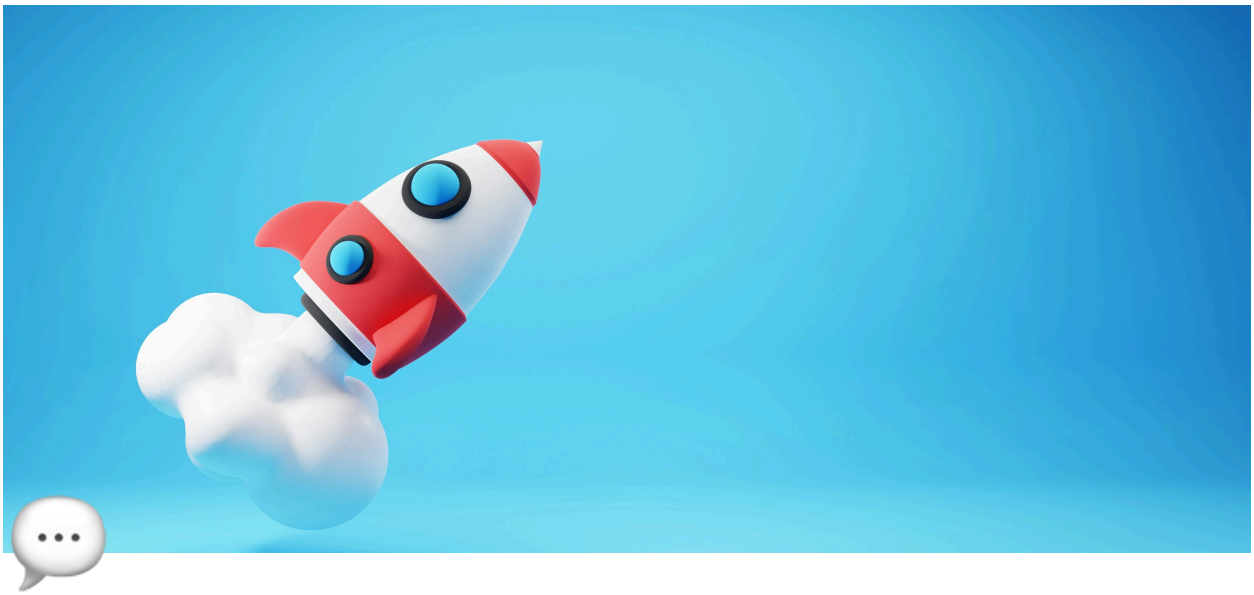


GSoC 2024 Proposal - Rocket.Chat AI Chat Conversation Thread Summarizer

About Me

Basic information

Name	Jeffrey Yu
Email	jeffreyzepengyu@g.ucla.edu
University	University of California, Los Angeles
Degree	Computer Science B.S.
GitHub	JeffreytheCoder
LinkedIn	Jeffrey Yu
Rocket.Chat username	jeffrey.yu
Timezone	PDT (UTC-7)

Self-introduction

I'm Jeffrey Yu, a final-year Computer Science student at the University of California, Los Angeles (UCLA). Born and raised in Shenzhen, China, I came to the United States

to pursue learning and working on software engineering. I love software because I enjoy spotting problems from people around me and building solutions that make their lives better. Particularly, I enjoy working on web apps and their intersection with technologies like generative AI and blockchain.

Prior experience

I have worked as a software engineer intern with four companies: Amazon, Paramount, Done., and TechFin.AI. I have built scalable applications used by thousands of people and worked on internal workflows that improve teams' productivity. From these experiences, I gained a strong foundation in full-stack development, especially with JavaScript / TypeScript, React, and NodeJS. In September, I'm going to join Meta as a software engineer and work under the Facebook News Feed team at Menlo Park, California.

Outside of work, I have built projects to solve problems for the UCLA student community, such as a club recruitment website called OnCampus and a dining hall review app called BruinEats. I love attending hackathons to build out new ideas and experiment with new technologies. These are some of my previous projects.

In addition, I love sharing knowledge and having discussions with the developer community. I wrote blogs about my projects and tutorials teaching people about full-stack development, which have 100k+ total views and are featured on "weekly must-reads" by Dev Community.

Why GSoC?

Although I have some industry experience, I have never and always wanted to get involved with open-source projects. I believe open-source is the best form of software development because it allows engineers to collaborate with full transparency and ultimate freedom. Any engineer, with any type of background, can participate and contribute to a project as long as it brings value, which is determined by the community collectively. This is the purest way of software development, and I want to be part of it. As a college student and an open-source beginner, GSoC is the perfect place for me to start.

Why Rocket.Chat?

When I was in high school, I started using Rocket.Chat during my internship at TechFin.AI. We hosted Rocket.Chat on our server to secure internal messages for

stock trading purposes. I got familiar with the platform and was fascinated by the customization capabilities.

So when I saw Rocket.Chat in the organization list of GSoC, I chose it without any hesitation. Besides having an amazing experience using Rocket.Chat, the tech stack fits perfectly with my skills. After familiarizing myself with the codebase and contributing, I found out the Rocket.Chat App ecosystem is so fascinating that I want to learn and contribute more with the opportunity of GSoC. Therefore, Rocket.Chat is the only one I chose among GSoC organizations, and this project is the only one for which I submitted a proposal.

Project

Basic information

Name	AI Chat Conversation Thread Summarizer
Mentors	José Petry, Devanshu Sharma
Project duration	90 hours

Overview

As generative AI becomes more integrated into people's lives, the Rocket.Chat community has decided to bring the power of AI into our communication platform. As part of the AI initiative in GSoC 2024, the AI Chat Conversation Thread Summarizer (referred to as the "AI Chat Summarizer" below) is an important project that would push forward the AI initiative and help grow the Rocket.Chat App ecosystem. This project aims to deliver a working Rocket.Chat App that can summarize any conversations using generative AI and provide valuable conversation insights to the users in seconds.

User benefits

We all get the annoying moments of scrolling through 80+ unread messages. With the AI chat summarizer, these headaches become the past. The new AI tool aims to quickly turn threads and messages into a concise summary, benefiting the Rocket.Chat users in the following ways:

1. **Boosted productivity**: Gartner reports that 47% of digital workers struggle to find the information they need, and 32% of employees have made the wrong decision due to a lack of awareness. The AI summarizer enables users to quickly

catch up on missed conversations, or revisit discussions to extract key points, aiding in faster information digestion and more informed decision-making. The summarizer keeps users focused on tasks that matter, reducing distractions from processing excessive information.

2. **Improved accessibility**: The summarizer ensures that all users, including those with a busy schedule or joining late, can easily access and understand the gist of discussions. This promotes inclusivity and ensures everyone has equal access to important insights, fostering a more connected community dialogue.
3. **Enhanced engagement**: Users are more likely to engage with and participate in conversations when they can quickly and easily understand the context and the key points. This leads to more vibrant, interactive communities in Rocket.Chat.

Analysis of Similar Products

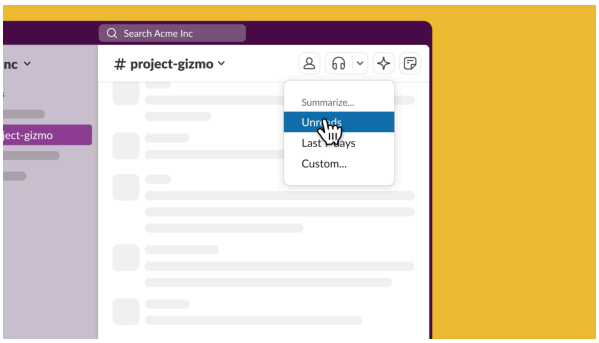
There are several similar products available in the market, and we can gain valuable insights by studying their features and design choices.

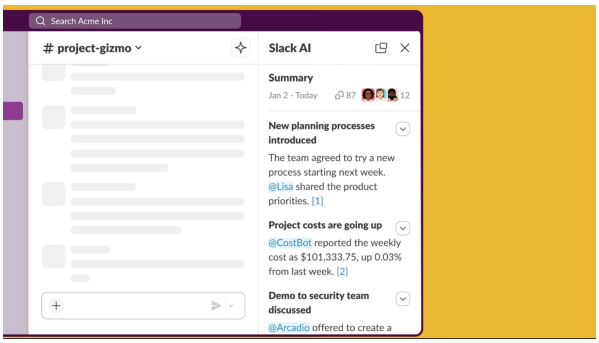
Slack AI

Slack AI, launched in February 2024, has three main capabilities:

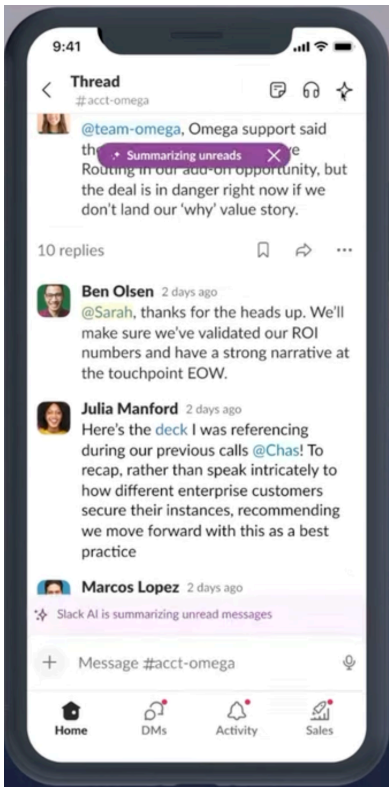
- **Search answers** that deliver personalized, intelligent responses to users' questions
- **Channel recaps** that generate key highlights from accessible channels
- **Thread summaries** that catch users up on long conversations in one click

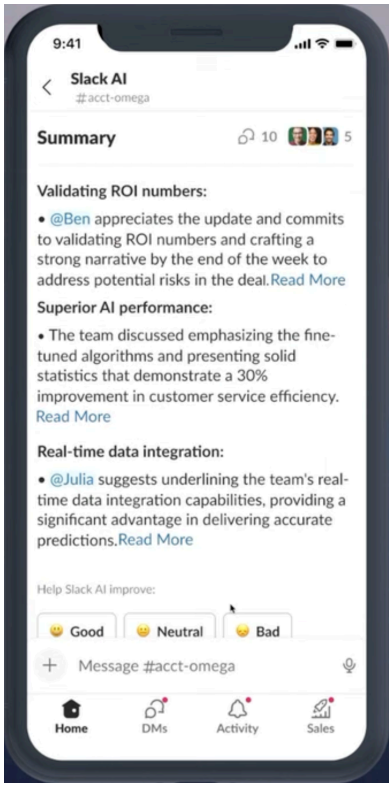
In channel recaps, users can get a quick rundown of the unread messages, summarize the past seven days, or set a custom date range.



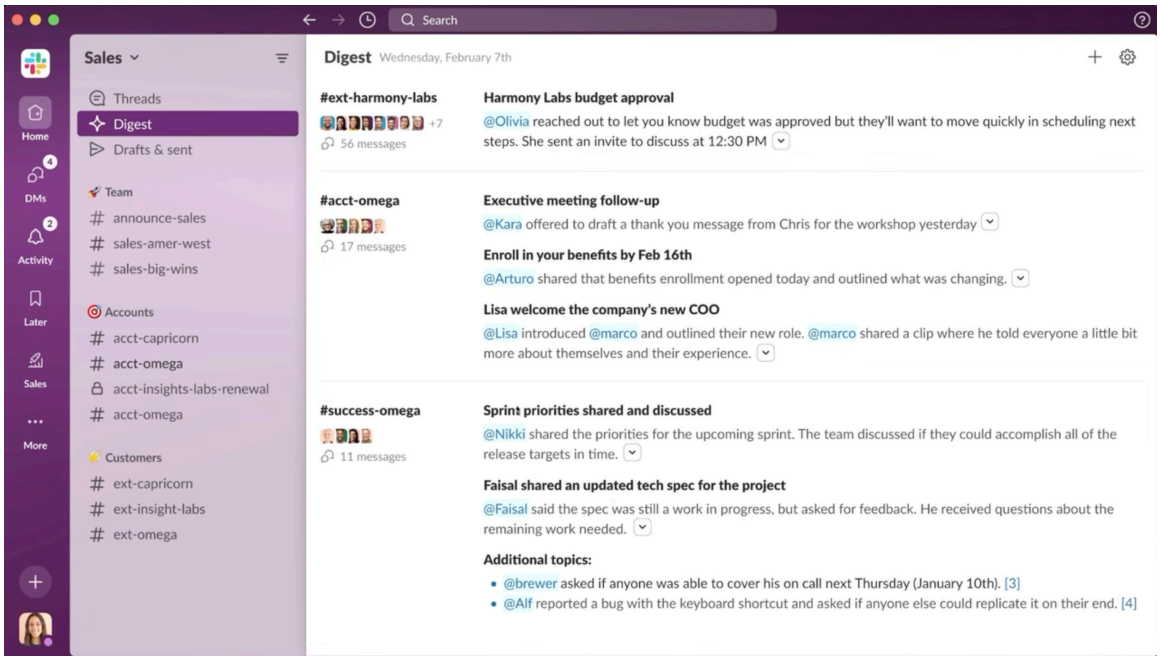


In any thread, users can click on the AI button, and it will generate a summary of the messages in the thread.





Slack AI offers personal daily digests, which summarize key points in each channel that are related to the current user.

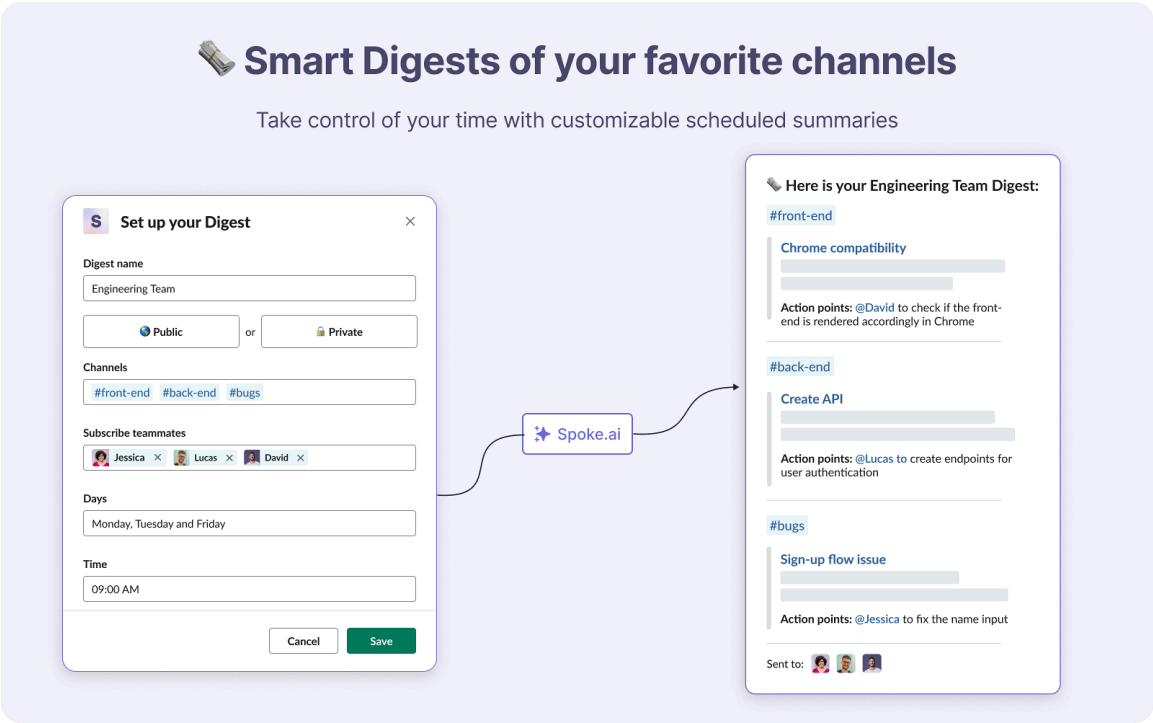


In addition, Slack promises not to share customer data with LLM providers or use customer data to train LLM. Slack makes sure that users' data remains in-house by hosting their LLMs directly on their infrastructure.

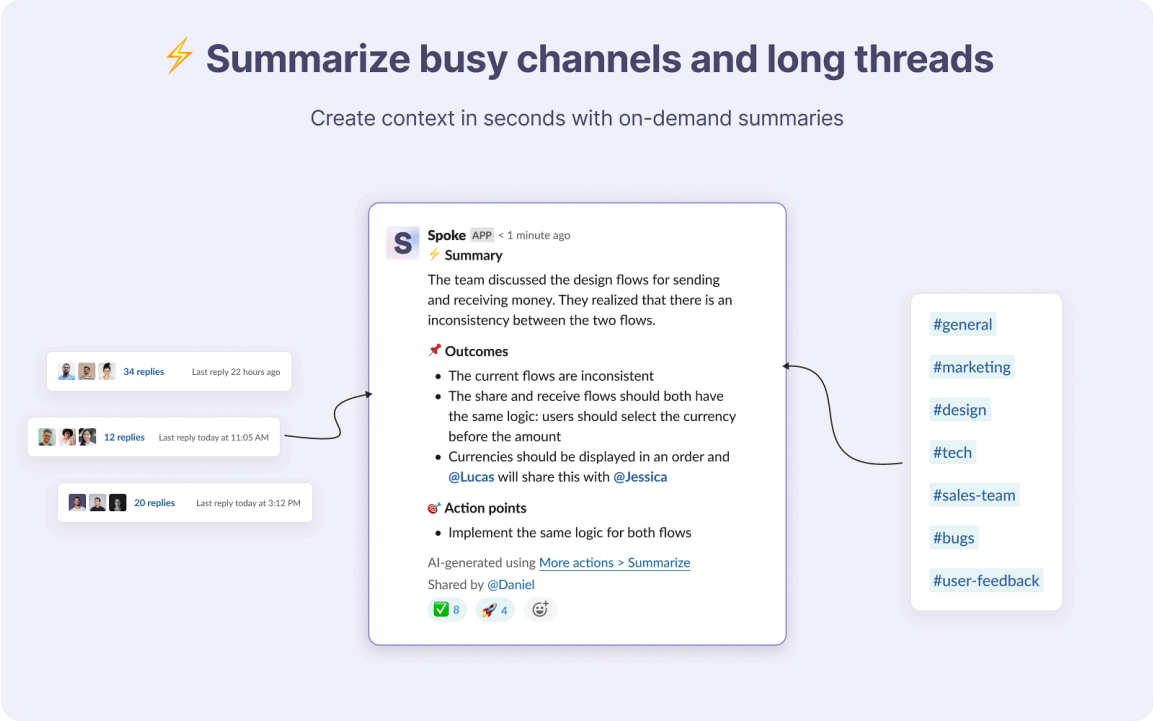
Currently, Slack AI is available as a paid add-on for Enterprise plans and is only available in English.

Spoke AI

Spoke AI is a Slack App that works as an AI assistant for Slack users. It can create AI digests that summarize up to 10 channels and share them with subscribed teammates on a scheduled time basis. The digest breaks down relevant topics by channel, shows who's involved, and triggers next-step actions like creating a Jira ticket or documenting something in Notion.



Moreover, Spoke AI can create a summarization of a thread or a channel. It can also turn the entire history of a channel into an AI assistant, who can answer any questions from the users.



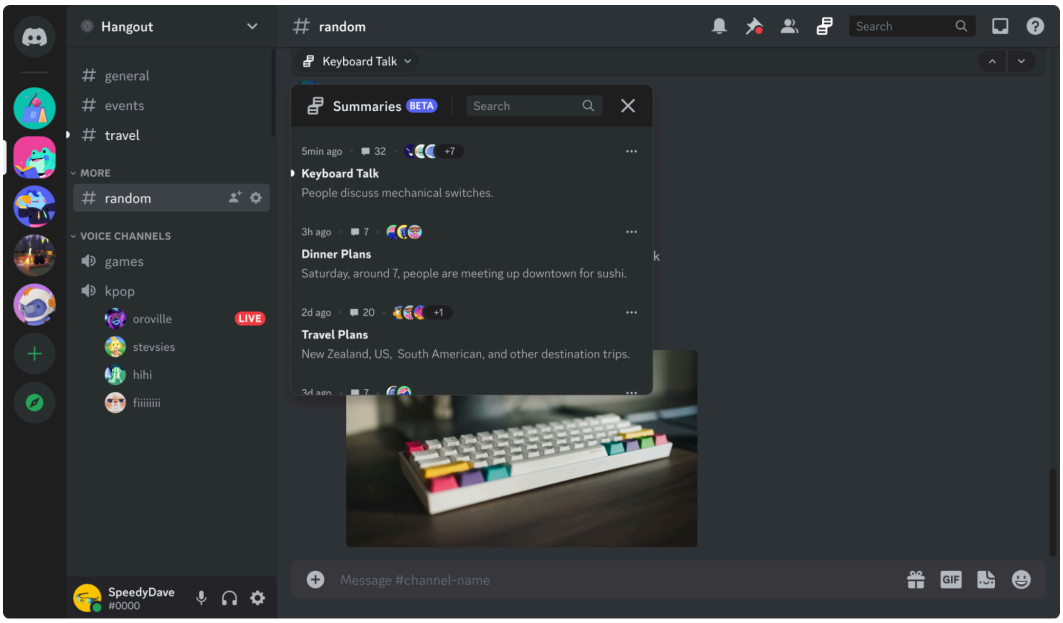
Last but not least, Spoke AI ensures privacy of users' data with the following features:

- **Limited data access:** Users can control which data Spoke AI can access
- **Automatic detection & anonymization** of all personal confidential data using proprietary methods
- **Full encryption** of all data at rest using AES-256 and in transit using HTTPS and RSA-2048
- **Optional incognito mode:** Users can use AI summarization without any of the input or output data being stored

Currently, the Slack App of Spoke AI allows free access for all Slack users. The Desktop App that works with Jira, Notion, Figma, etc. is free during beta and is planned to charge $29 per user per month. Lastly, it requires an enterprise plan with custom pricing to plug Spoke AI into users' own LLMs and physically keep data on users' servers.

Discord In-channel conversation summaries

Discord introduced its In-channel conversation summaries feature in November 2023. The feature consolidates messages the user receives into easy-to-navigate topics. For each topic, the user can see a conversation preview, the number of messages about a specific topic, and who is part of the discussion.



Currently, this feature is in beta mode and limited to a few Discord servers.

Deliverables

During the GSoC period, I would focus on delivering the following features for the AI Chat Summarizer App:

Slash commands

In a thread, the user can use the command:

- ***/summary*** - To generate a summary of the thread (the App will send a private message to the user)

In a channel, the user can use commands:

- ***/summary unread*** - To generate a summary of the user's unread messages in the channel
- ***/summary [date-range]*** - To generate a summary of the messages in the channel during a certain date range
    - *[date-range]* - Common date ranges like today, yesterday, this week, or custom date ranges by the user
- ***/summary*** - To generate a summary of all messages in the channel
    - Not recommended because the channel can potentially have a large number of messages

    - The App will pop up a modal if the message count in the channel exceeds a certain threshold

For any command above, the user can add *ask* and use the command:

- ***/summary [option] ask [question]*** - To ask AI a question regarding the messages selected from *[option]*
    - *[option]* - If in a channel, it can be empty, *unread*, or *[date-range]*. If in a thread, it is ignored and refers to all messages in the thread.
    - For example, */summary unread ask What did @Dnouv talk about in the channel?*

Finally, the user can use this command anywhere in Rocket.Chat:

- ***/summary schedule*** - To schedule a personal digest that includes summaries of the selected channels
    - In a pop-up modal, the user can select the channels and choose a scheduled time
    - The personal digest will be sent to the user's DM with the App

App settings

The App settings page would include the following configurations, some of which will be introduced in the following sections:

- *model_api* - The user can plug in their model by using the model's API endpoint
- *model_request* - Format of the model's API request, including method, headers, content, etc.
- *model_response* - Format of the model's API response
- *prompt* - The user can customize the prompt used to generate the summary
- *personal_digest_channels* - Channels selected for personal digest
- *personal_digest_schedule* - Personal digest schedule (in CronJob format)
- *hard_token_limit* - The hard limit on the token size of messages input for summary

Prompt Engineering

Considering the large number of messages that can exist in a thread or in a channel, the AI model could potentially output a long summary that covers many details in the

conversations. It's important that we keep the summary concise and straight to the point, allowing the users to understand conversations quickly and easily by looking at the high-quality summary. To achieve this, we need to find the best prompt for guiding the AI model.

Therefore, I conducted a prompt engineering study, which is documented in this Jupyter Notebook. I evaluated how different prompts perform under our specific use case: **summarizing online dialogues in a concise way**. The following sections will introduce the main components of the study and go through how I carried out the prompt engineering procedure.

SAMSum Corpus

SAMSum Corpus is a human-annotated dialogue dataset for abstractive summarization. SAMSum is the perfect dataset for our task because it contains 16k+ messenger-like conversations with summaries. It was prepared by Samsung R&D Institute Poland and is distributed for research purposes.

Conversations in SAMSum were created and written down by linguists fluent in English. Linguists were asked to create conversations similar to those they write on a daily basis, reflecting the proportion of topics of their real-life messenger conversations. The style and register are diversified - conversations could be informal, semi-formal, or formal, and they may contain slang words, emoticons, and typos. Below is an example dialogue with an annotated summary:

Dialogue
Blair: Remember we are seeing the wedding planner after work Chuck: Sure, where are we meeting her? Blair: At Nonna Rita's Chuck: Can I order their seafood tagliatelle or are we just having coffee with her? I've been dreaming about it since we went there last month Blair: Haha sure why not Chuck: Well we both remmber the spaghetti pomodoro disaster from our last meeting with Diane Blair: Omg hahaha it was all over her white blouse Chuck: :D Blair: :P
Summary
Blair and Chuck are going to meet the wedding planner after work at Nonna Rita's. The tagliatelle served at Nonna Rita's are very good.

Table 2: Example of a dialogue from the collected corpus

Source: ACL Anthology

SAMSum is available on Hugging Face so we can load the dataset like this:

```
from datasets import load_dataset
dataset = load_dataset("samsum", split='train')
```

ROUGE-L metric

ROUGE (Recall Oriented Understudy for Gisting Evaluation) is a metric primarily used for comparing a computer-generated summary of text with a reference (e.g. human-generated) summary. It's mostly used for evaluating text-summarization tasks in NLP and LLM studies. The value of ROUGE metrics, ranging from 0 to 1, indicates the similarity between the computer-generated summary and the reference summary.

In this study, I use ROGUE-L, a sub-metric of ROUGE that evaluates summary quality by measuring the **longest common subsequence (LCS)** between the generated and reference summary. ROUGE-L can better capture the overall structure and flow of the

summaries than other sub-metrics like ROUGE-1 or ROUGE-2. We can use *load* from Hugging Face to define a helper function that calculates ROUGE-L:

```
rouge_l = load("rouge")
def calculate_rouge_l(reference, generated):
    results = rouge_l.compute(
      predictions=[generated],
      references=[reference]
    )
    return results["rougeL"]
```

Prompt evaluation

To evaluate a prompt, we randomly choose 30 samples from the SAMSum dataset. The sample size of 30 is a sufficiently large sample size by the Central Limit Theorem (CLT) and ensures our results are statistically sound.

Each SAMSum sample has a dialogue and a reference summary. For each of the samples, we use OpenAI GPT-3.5 Turbo to generate a summary from the dialogue and compare it with the reference summary by calculating the ROUGE-L score. I chose GPT-3.5 Turbo for this study because it's a widely available, common LLM to the general public. This makes sure that our ROUGE-L score calculations serve as a reliable benchmark for commonly used models.

At the end, we take the mean of all 30 ROUGE-L scores. The mean reflects how well the prompt is performing in summarizing the dialogue.

```python
def evaluate_prompt(prompt, instruction):
    scores = []
    for example in np.random.choice(dataset, size=30, replace=False):
        formatted_prompt = prompt.format(
            dialogue=example["dialogue"],
            instruction=instruction
        )
        api_response = call_openai_api(formatted_prompt)

        score = calculate_rouge_l(example["summary"], api_response)
        scores.append(score)

    average_score = np.mean(scores)
    print(f"Average ROUGE-L score: {average_score}")
```

Prompt Iteration

We start from a baseline prompt, then gradually apply prompt engineering techniques, and see if they improve the prompt by evaluation. By iterating our prompt to better versions, we can get a high-quality prompt at the end.

Zero-shot prompt

First of all, we have a zero-shot prompt as the baseline prompt:

```
Summarize the following dialogue.

Dialogue: ###
{dialogue}
###

Summary: """
```

The baseline prompt has one-line instruction with no examples. The prompt allows templating, where the actual dialogue can be filled into *{dialogue}*. I added the *###* lines to distinguish dialogue from instruction and summary, which is a good practice suggested by OpenAI. The model will generate the summary by completing the text after *Summary:*.

After evaluation, the prompt gets a ROUGE-L score of 0.289. According to Klu.ai, a ROUGE-L score of 0.3 to 0.4 is considered moderate, and a score around 0.4 is considered good. Thus, our baseline prompt doesn't reach moderate.

Zero-shot prompt with polished instruction

When I looked at some of the generated summaries from the baseline prompt that received a low ROUGE-L score, I found that they are much longer and cover too many details compared to the reference summary. For examples:

```
Referenced summary: Rice will visit Bert.
Generated summary: Rice asks permission to come over, Bert agrees,
  and Rice confirms they are on their way.
ROUGE-L score: 0.2

Referenced summary: Rosie is going to lend Carl 10 dollars.
Generated summary: Carl asks if anyone can lend him $10 until Friday,
  but when Newt declines, Rosie agrees to lend him the money
  and asks for his account number. Carl expresses gratitude towards Rosie.
ROUGE-L score: 0.2
```

While the generated summaries include details that are missed in the reference summaries, our goal is to create summaries that **concisely** capture the main points, so ignoring some details is acceptable as part of the trade-off for conciseness. To improve the conciseness in generated summaries, I polished the instruction to:

```
Summarize the following dialogue using 1-3 short and simple sentences.
Use as fewer words as possible. Mention the names of specific persons.
```

In the new instruction, I included specific and detailed demands about conciseness, like *1-3 sentences* and *fewer words*. This follows one of the good practices suggested by OpenAI, which could improve the accuracy of the prompt.

Besides, I included the instruction to mention specific names in the prompt because I noticed that some summaries replace names with pronouns or general terms like "friends." However, it's important to mention specific Rocket.Chat users in the

summaries generated from our App. This ensures clarity and helps our App users understand exactly who is involved in the summaries.

After evaluation, the prompt gets a ROUGE-L score of 0.294, which is a slight improvement from our baseline prompt.

Few-shot prompt

To further improve the summarization accuracy of our prompt, we can use the few-shot technique by including a few examples of dialogue + summary from the SAMSum dataset. By showing the model examples of how to summarize dialogue in a concise way, we expect the model to output better summaries. To test the improvement of performance by changing to a few-shot, I changed the instruction back to the baseline one:

```
Summarize the following dialogue.

Dialogue: ###
Tim: Hi, what's up?
Kim: Bad mood tbh, I was going to do lots of stuff but ended up procrastinating
...
###

Summary: Kim may try the pomodoro technique recommended by Tim to get more stuff done.

---

Summarize the following dialogue.

Dialogue: ###
John: Ave. Was there any homework for tomorrow?
Cassandra: hello :D Of course, as always
...
###

Summary: John didn't show up for class due to some work issues with his boss.
  Cassandra, his teacher told him which exercises to do, and which chapter to study.
  They are going to meet up for a beer sometime this week after class.

---

Summarize the following dialogue.

Dialogue: ###
Leon: did you find the job yet?
Arthur: no bro, still unemployed :D
...
###

Summary: Arthur is still unemployed.
  Leon sends him a job offer for junior project manager position.
  Arthur is interested.

---

Summarize the following dialogue.

Dialogue: ###
{dialogue}
###

Summary:
```

After evaluation, the prompt gets a ROUGE-L score of 0.312, making the performance in the moderate range.

Few-shot prompt with polished instruction

Finally, I changed the instruction in the few-shot prompt to the polished one we have. After evaluation, the prompt gets a ROUGE-L score of 0.386. The performance of this prompt improves significantly from the previous ones and jumps into the good range. This proves our prompt engineering techniques, few-shot prompt and polished instruction, to be useful.

After the project starts, I will explore more prompt engineering techniques and further iterate our prompt. In addition, I will conduct a different prompt engineering study for command */summary ask*, aiming to find a prompt that outputs high-quality Q&A responses from taking the user's questions.

Llama 2 results

Since the users can connect our App with any model, we want to test the prompt performance across different models. Besides OpenAI's GPT, another widely used model is Meta's Llama 2. Considering most Rocket.Chat users favor open-source software, Llama 2 could be a popular choice. Therefore, I conducted a similar study with Llama 2, and below are its performances with different prompts:

Prompt	ROUGE-L score
Zero-shot prompt	0.244
Zero-shot polished prompt	0.324
Few-shot prompt	0.299
Few-shot polished prompt	0.326

As shown in the table, changing from a zero-shot to a few-shot prompt and having polished instruction improves performance, which is the same as the findings from the GPT study. As the project moves forward, I will compare prompts across more models and choose the one that has the best-generalized performance. In addition, we can track which models are used in our App and get insights to improve our prompt.

Handling token limits

Considering the length of our few-shot prompt and the large number of messages, our total prompt might exceed the token limit of the models. After calculation, our finalized prompt has 720 tokens. However, common models like Llama 2 and GPT-3.5 Turbo has a token limit of 4096 tokens. Therefore, we need to come up with solutions to avoid exceeding the token limit.

The first approach is to adjust our prompt based on the token limit, which is commonly provided in the model API response. However, downgrading our few-shot prompt to one-shot could drop the ROUGE-L score by ~0.1, as shown in our study. One solution is to keep a few-shot prompt, but compare the performance of summary examples in different lengths, and choose the shortest few-short prompt that performs the best. I will conduct a follow-up study for this.

Moreover, for messages that surpass the token limit, we can apply the chunking technique. We break down messages into small chunks, generate a summary for each chunk, and synthesize a final summary from the individual summaries.

In addition, it's good to have a hard limit on the total message size. While the previous strategies focus on dealing with soft limits, a hard limit allows our App to decline a request if the user is trying to summarize a lot of messages. This saves the Rocket.Chat server and the user's model from processing unmanageable load. The user can update the *hard_token_limit* setting to adjust the limit according to their server or model capability with caution.

App Implementation

To verify the viability of the project, I built a prototype of the AI Chat Summarizer App. Below I will explain some parts of the prototype code and how I plan to implement planned features in the *Deliverables* section.

/summary (in thread)

To summarize all messages in a thread, we need to:

1. Get all messages in the current thread (the user must have opened the thread in the sidebar) by using *IThreadRead.getThreadbyId(id)*. method from App Engine

```
private async getThreadMessages(...):
    const threadReader = read.getThreadReader();
    const thread = await threadReader.getThreadById(threadId);

    // Format messages as username: messageText
    const messageTexts: string[] = [];
    for (const message of thread) {
        if (message.text) {
            messageTexts.push(`${message.sender.name}: ${message.text}`);
        }
    }
    return messageTexts;
}
```

2. Feed the messages to the AI model and ask it to return a summary as defined by the prompt

```
private async summarizeMessages(...) {
    const body = {
        messages: [{
            role: "system",
            content: createPrompt(messages)
        }],
        // ... Follow model_request format
    };
    const response = await http.post(MODEL_API, {
            headers: {
            Authorization: `Bearer ${MODEL_API_KEY}`,
            "Content-Type": "application/json",
        },
        content: JSON.stringify(body)
    });

    // With parsing that follows model_response format
    return JSON.parse(response.content);
}
```

3. Send the summary as a private message to the user

```
private async notifyMessage(...) {
    const notifier = read.getNotifier();

    const messageBuilder = notifier.getMessageBuilder();
    messageBuilder.setText(message);
    messageBuilder.setRoom(room);

    if (threadId) {
        messageBuilder.setThreadId(threadId);
    }

    return notifier.notifyUser(user, messageBuilder.getMessage());
}
```

Check out the demo of */summary* used in a thread:

https://youtu.be/9bNEUDrNLRo

/summary (in channel)

Creating a summary of messages in a channel is similar to that for a thread, except for the method to get messages. While we use *IThreadRead.getThreadbyId(id)* to get messages in a thread, we use *IRoomRead.getMessages(roomId)* method to get messages in a channel, which is currently not implemented in the Apps Engine. Therefore, we need to clear out this major roadblock before we can implement channel summarization.

Previous attempts of *getMessages*

Back in 2019, there was an attempt to implement the *getMessages* method in Apps Engine PR #162 and its corresponding Rocket.Chat part, PR #17605. The attempted solution tries to query all visible messages within a room using a MongoDB cursor. However, considering the large number of messages in a channel, this approach could potentially lock a server whenever an App calls *getMessages*, as mentioned by @graywolf336. He closed the PR and suggested an alternative approach, which is to implement a query builder with pagination support that is database agnostic in the style of the Apps Engine.

In 2023, @Douv implemented another approach in his Apps Engine fork and the Rocket.Chat counterpart. The attempted solution added MongoDB query options,

including *limit*, *skip*, *fields*, and *sort*. The implementation sets the max limit to 100 to restrict the number of queried messages per method call. This approach solves the server load problem, but constrains the flexibility of the caller Apps in getting the messages they need. For example, our */summary [date-range]* command wants to get messages from December last year. It would be challenging for the App to precisely locate that range of messages using the four given options. Therefore, we need a more flexible approach for the *getMessages* implementation.

Proposed solution of *getMessages*

One potential solution is to enable more filters as parameters to *getMessages* like this:

```
getMessages(startDate, endDate, username, unread, ...)
```

For example, if an App wants to get messages from a certain username, it can call *getMessages(username="jeffrey.yu")*. The *getMessage* method implementation will map filter parameters to underlying query operators in MongoDB cursor. For example, matching a username would be mapped to:

```
db.messages.find({ username: { $eq: "jeffrey.yu" }})
```

To avoid the parameter overload issue as filters increase, we can implement a query builder, in the same way as @graywolf336 suggested:

```
function getMessages(query) { … }
const query = messagesQuery()
  .startDate('2024-03-01')
  .endDate('2024-03-31')
  .username('jeffrey.yu')
  .unread(true)
  .find();
```

The message filtering would have a good time performance using MongoDB indexing, considering searching messages within a channel gives results quickly. To solve the server load issue of having a large number of filtered results, we can provide a paginated version of the function:

```
getMessagePaginated(..., pageSize, pageNumber)
```

pageSize maps to the underlying *limit* and *skip* options in MongoDB cursor, and we can set a max limit of *pageSize*. The user can navigate through the pages by incrementing *pageNumber*.

/summary ask

Like */summary*, */summary [option] ask [question]* follows a similar implementation:

1. Get messages in the current thread / channel selected by *[option]*
2. Feed the messages to the AI model and ask it to answer *[question]* as defined by the prompt
    a. I will find the best Q&A prompt by conducting a study similar to how we got the best prompt for summarization in the *Prompt Engineering* section
3. Send the summary as a private message to the user

/summary schedule

There are two settings for scheduling a personal digest: channels and scheduled time. The user can either modify the settings directly in the settings page, or call */summary schedule* and modify the settings to show a pop-up modal. Here are the steps to implement this feature:

1. Create settings. The scheduled time is in CronJob format, but in case the user is not familiar with CronJob, we can visualize the time schedule by using a series of UiKit input elements in the pop-up modal.

```
export const settings: ISetting[] = [
    {
        id: AppSettings.PERSONAL_DIGEST_CHANNELS,
        i18nLabel: 'PersonalDigestChannels',
        type: SettingType.MULTI_SELECT,
        required: false,
        public: false,
        packageValue: [],
    section: "PersonalDigest",
    },
    {
        id: AppSettings.PERSONAL_DIGEST_SCHEDULE,
        i18nLabel: 'personalDigestSchedule',
        type: SettingType.STRING,
        required: false,
        public: false,
        packageValue: "0 9 * * *", // default: 9AM every day
    section: "PersonalDigest",
    }
];
```

2. Extend App configuration with the settings and register a processor for the personal digest scheduler.

```
public async extendConfiguration(...) {
    // Provide settings
    await Promise.all(
        settings.map((setting) ⇒
            configuration.settings.provideSetting(setting);
        )
    );

    // Get subscribed channels from setting
    const channels: string[] = await this.getAccessors()
        .environmentReader.getSettings()
        .getValueById(AppSettings.PersonalDigestChannels);

    // Register scheduler processor
    configuration.scheduler.registerProcessors([
        {
            id: ProcessorsEnum.PERSONAL_DIGEST,
            processor: async (jobData, ...) ⇒ {
                jobData["channels"] = channels;
                await sendPersonalDigest(jobData, ...);
            },
            startupSetting: {
                type: StartupType.RECURRING,
                interval: "0 9 * * *", // default at startup
            },
        },
    ]);
}
```

3. When reaching the scheduled time, the scheduler process will trigger *sendPersonalDigest*. This method will summarize messages in the selected channels since the last time the scheduler was called. For example, if the scheduled time is 9 AM every day, the method will summarize messages from 9 AM yesterday till now (9 AM today). Then, the method will combine the summaries into one message and send it to the user via DM.

```
export async function sendPersonalDigest(jobData, ...) {
  const { channels } = jobData;

  // Map channel name to its summary
  const channelSummaries = {}
  channels.forEach(async channel => {
    const channelName = await getChannelName(channel)
    channelSummaries[channelName] = ''
  })

  // Get channel summaries
  await Promise.all(channels.map(async (channel) => {
    const channelSummary = await getChannelSummary(channel)
    channelSummaries.push(channelSummary)
  }))

  // Combine summaries into one message
  const digestMessage = createDigestMessage(channelSummaries)
  await notifyMessage(..., digestMessage)
}
```

4. On setting page update or when the user calls */summary schedule*, we update the scheduler to make sure the new schedule is enforced

```
public async onSettingUpdated(...) {
    const schedule: string = await this.getAccessors()
        .environmentReader.getSettings()
        .getValueById(AppSettings.PersonalDigestSchedule);

    // Cancel the previous scheduler job to avoid duplicate
    await configurationModify.scheduler.cancelJob(
        ProcessorsEnum.PERSONAL_DIGEST
    );

    await configurationModify.scheduler.scheduleRecurring({
        id: ProcessorsEnum.PERSONAL_DIGEST,
        interval: schedule,
    });
}
```

My Contributions

When I stepped into the Rocket.Chat community, I was quickly attracted by some interesting projects and started making contributions to them. As of March 31, I have opened 15 issues, created 10 PRs, and all of my PRs are merged. I rank #3 out of 260 contributors on the GSoC Contribution Leaderboard.

Here is a summary of my contributions:

My Rocket.Chat Contributions

Aa Pull Request	Repository	Merged
Align invitation sidebar UI with RocketChat	EmbeddedChat	☑
Add tooltips to message toolbox	EmbeddedChat	☑
Add download function and file name to messages attachments	EmbeddedChat	☑
Add error message for wrong PR number	Apps.Github22	☑
Change notify room to notify user for individual commands	Apps.Whiteboard	☑
Make search tab consistent with threads & RC	EmbeddedChat	☑
Change starred and pinned message windows to sidebar	EmbeddedChat	☑
Refactor sidebar components	EmbeddedChat	☑
[Enhance] Improve readability of comment created time	Apps.Notion	☑
[FIX] Starred & Pinned messages sidebars fetch APIs infinitely	EmbeddedChat	☑

From contributing to EmbeddedChat, a simplified version of Rocket.Chat, I understood the core functionalities of Rocket.Chat, including APIs and frontend features. By contributing to the Rocket.Chat Apps, I learned App development and operating with the Apps Engine interface, which helped me in building the prototype of AI Chat Summarizer. These experiences allow me to quickly onboard the project and deliver features for the AI Chat Summarizer once GSoC starts. I believe GSoC is a jump start for me to get into the Rocket.Chat community, and I plan to continue my contributions moving forward.

Timeline

Pre-GSoC

April 2-30

- Keep contributing to Rocket.Chat projects
- Iterate summarization prompt by experimenting with chunking techniques and testing with different models
- Conduct another prompt engineering study for Q&A prompt

Community bonding period

May 1-26

- Meet mentors and interact with the Rocket.Chat community
- Discuss with mentors and finalize the project design
- Keep working on items in the Pre-GSoC section

Week 1-4 (Coding period starts)

May 27 - June 23

- Set up project, including npm, ESLint, Prettier, i18n, GitHub Actions, etc.
- Implement */summary* command in thread
- Work on *getMessages* implementation in Apps Engine

Week 5-8

June 24 - July 21

- Implement */summary* command in channels (once the *getMessages* blocker is removed)
- Implement */summary ask* command
- Iterate summarization and Q&A prompts from the actual results of running implemented commands
- Receive feedbacks from midterm evaluation and improve accordingly

Week 9-12

July 22 - August 18

- Implement */summary schedule* command

- Perform end-to-end tests on different AI models and user environments. Identify and address the discovered bugs and issues
- Write documentations, issue & PR templates, contribution guidelines, etc. to prepare for community contributions post-GSoC
- ...and some buffer time to finish uncompleted tasks

Final week

August 19-25

- Write a project summary and prepare for the presentation on Demo Day
- Receive final feedbacks from mentors and plan future roadmap for the App

FAQs

Do you have time for GSoC after graduation?

Yes! My job at Meta starts in September, which is after GSoC ends. I want to dedicate my summer time for GSoC.

What is your time commitment for this project?

I don't have any other commitments during the summer, so I can work full-time and dedicate 40 hours (or more) per week on the project.

How do you plan to communicate with mentors?

I'm usually available between 9AM to 12AM PDT for any kind of communication. I can adjust my working time for meetings across timezones.

Why do you always use dark mode?

Because light attacts bugs :)