# Podcast recommendation system with LLM and RAG

Lance Kind
AI Product Developer

Tim Zack
Content Director

Dave Prior
Podcast Property Creator

Josh Fortune
AI Product Developer

## Abstract

Audio podcasts have grown to have thousands of episodes and keyword search isn't very effective at discovering what listeners are interested in. Creating a concierge that can understand semantically what the listener is seeking will be a vast improvement over search. We'll describe a software architecture which we will deploy to several clients using pre-trained LLM's and embedding as a convenient and simple way to bring listeners to content they care about. In this paper we will go through the learnings that lead us to a software architecture for implementing a concierge chatbot for podcast episodes.

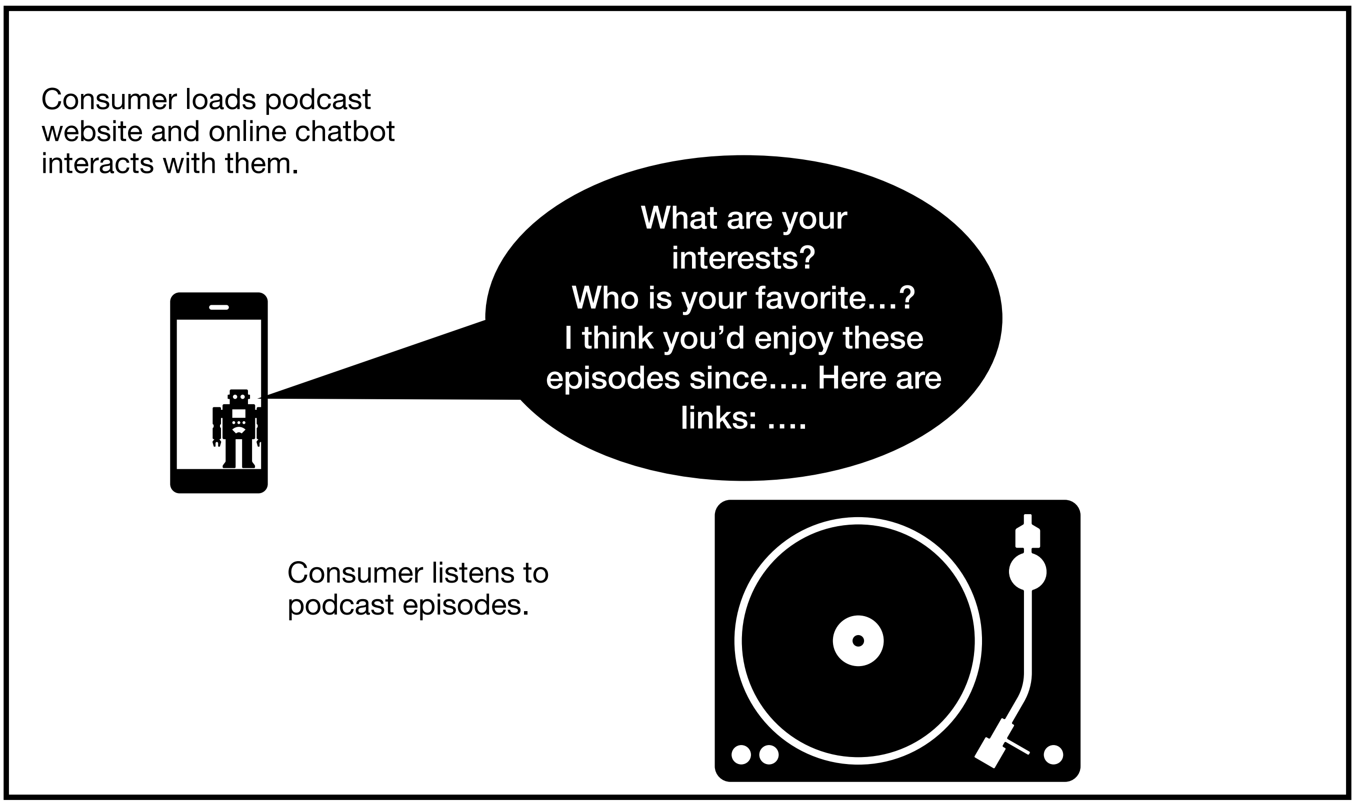
## Introduction

For this project, we're using a Transfer Learning approach in our podcast chatbot project utilizing a pre-trained Language Model (LLM), such as GPT-3.5, to harness its extensive language understanding capabilities. Our focus involves fine-tuning the model using a specific corpus of podcast episode data through embedding generation. This process involves extracting meaningful embeddings from the podcast transcript, allowing the model to provide nuanced and context-aware recommendations to users based on the information encoded in the embeddings.

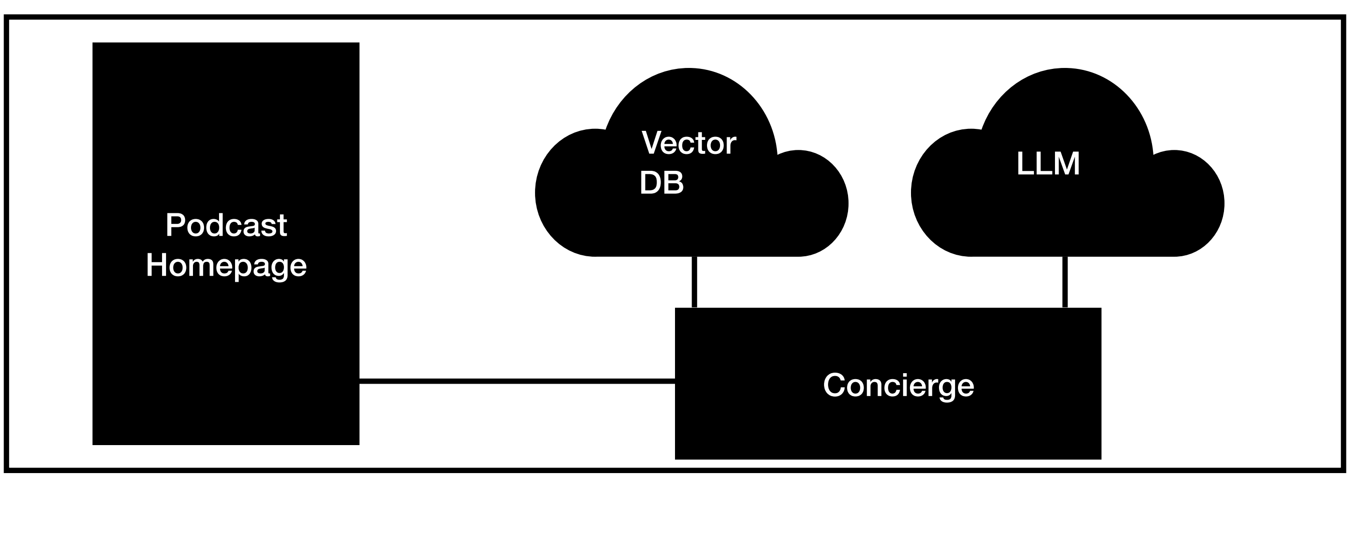
Regarding the tool chain, as this is using a corpus of public accessible information, there aren't concerns about having to do embedding generation in a private manner which leaves the solution space wide open. The concierge should be able to do the following:

- handle multilingual conversations
- Guide visitors to podcast episodes that would be of interest, mentioning episode number and short summary, and a link to the episode.

Architecture context



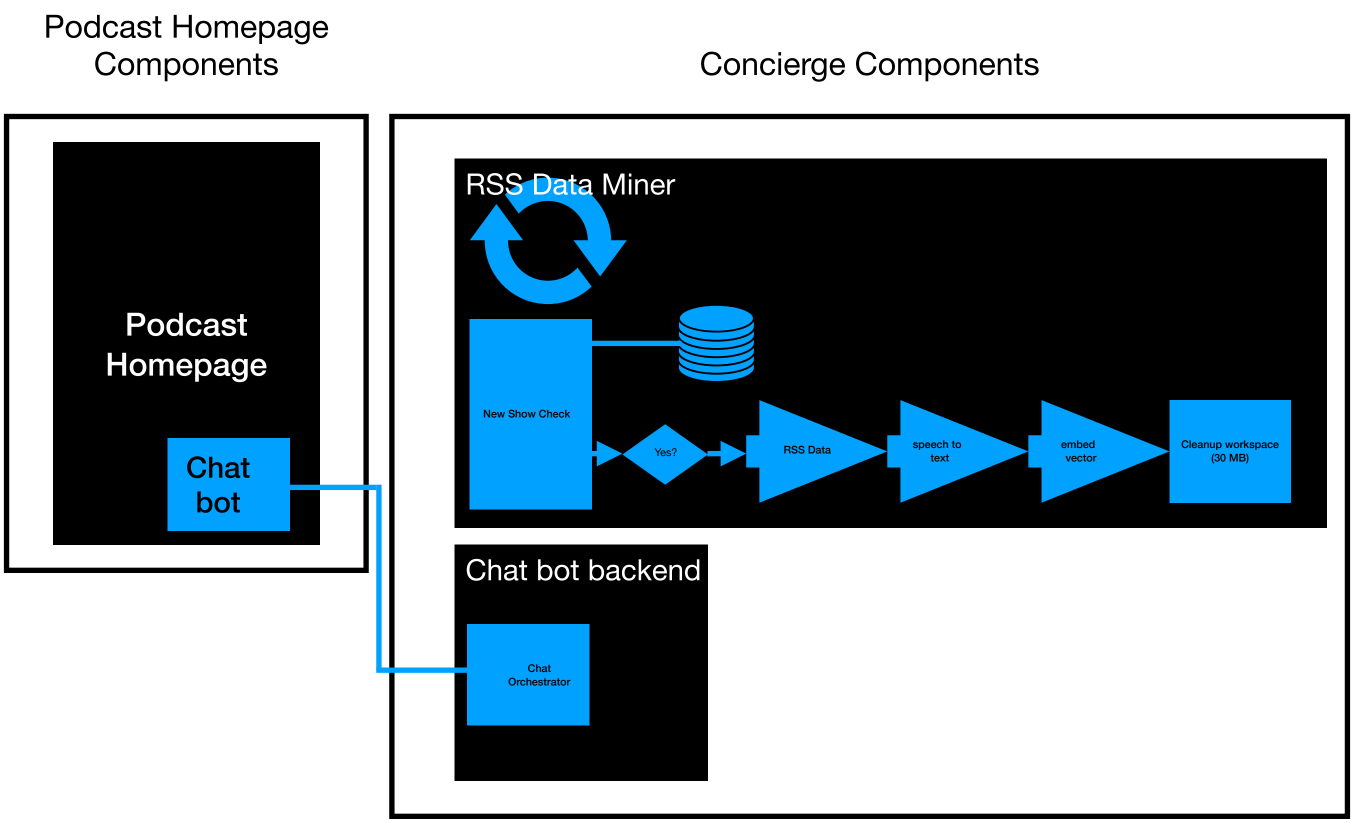

Architecture Containers



Experiments were conducted to discover the following unknowns:

1. how to define data features and structure so that the concierge directs the user to interesting episodes
2. what tool chain will extracts from audio semantic data
3. what tool chain will create the embedding in an automated way
4. discover method to update the semantic data used by the concierge on a weekly bases as new podcast episodes come out

Since podcasts announce episodes and metadata through RSS (Really Simple Syndication) feeds, a likely implementation for bullet 4 will be an RSS Data Miner that is stateless with the exception of a database, and so it would use a serverless technology. The database will have state to know if it has already consumed the latest RSS feed. The Chat bot backend will have some ephemeral local state (conversation memory) and would use a container.
Right now, it's an open question as to whether the implementation will: use a relational database or leverage state from the vector database. For frequent transactions, a relational database is expected to use less resources than querying the vector DB via the LLM. However since podcasts are often only weekly or bi-weekly, it is believed that leveraging the vector DB will actually require less resources than a relational database. And there is also the possibility of directly querying the vector database without the LLM which will reduce resources again.




# Data pipeline

Converting MP3 podcast data into a structured dataset for embedding generation involves several steps. Here's a breakdown of the process:

1. **Audio-to-Text Transcription:**
   - Use an automatic speech recognition (ASR) system or transcription service to convert the spoken content in MP3 files into textual data. This step creates a transcript of the podcast episodes.

2. **Text Preprocessing:**
   - Clean and preprocess the transcribed text to remove any irrelevant information, correct errors, and ensure consistency. This step may involve handling punctuation, special characters, and normalizing text.

3. **Structuring Data into Columns:**
   - Organize the transcribed text into structured data. Create columns for relevant information such as episode numbers, titles, and summaries. Each row in the dataset corresponds to a specific podcast episode.

4. **Feature Engineering:**
   - If available, incorporate additional features such as timestamps, speaker information, or episode descriptions. These features will enrich the dataset and provide more context for embedding generation.

5. **Data Format:**
   - Save the structured data in a suitable format, such as CSV or JSON. This format ensures that the data can be easily loaded and processed by tools and libraries used for embedding generation.

6. **Tokenization:**
   - Tokenize the textual data to break it down into individual units, such as words or subwords. Tokenization is a crucial step before applying embedding models.

7. **Embedding Generation:**
   - Use a pre-trained embedding model or embedding generation technique (e.g., Word2Vec, FastText, or BERT) to transform the tokenized text into embeddings. These embeddings capture semantic information from the podcast episodes.

8. **Normalization (Optional):**
   - Depending on the embedding model used, you may consider normalizing the embeddings to ensure consistent scales across different dimensions.

9. **Storage:**
   - Save the generated embeddings along with the structured data. This step is crucial for later retrieval and use in building your chatbot.

By following these steps, the MP3 podcast data will be organized into a structured dataset suitable for embedding generation. This structured dataset can then serve as a valuable resource for an LLM chatbot to provide recommendations based on the content of the podcast episodes.

# 1. Audio to Text Transcription

OpenAI Whisperer.  and needs to handle up to 1 hour.
https://www.assemblyai.com Is a $0.37 per hour alternative.  The metadata from the RSS feed for the episode will be used to prompt Whisper to handle spelling of host, guest, and company names.

# 2. Text Preprocessing

# 3. Structure podcast data

With the advent of LLMs' ability to process natural language, this raises questions on if there is an advantage to using different structures. Ideally, we can use whatever best fits in our data pipeline with least effort. The two choices we tested are using **natural language structure** or using **columnar structure**.

<pull in text about features and data.>

Regardless of which structure, [1, LLM + Spreadsheets, section “What do I do with all these columns of data?”] suggests that semantically rich content should be vectorized along with "related content" so that "related content" is associated with the semantic data. So in Table 2’s case, Summary is the semantic rich content to be vectorized, and we want the LLM to use the other columns (Series name, Episode name, and Link to series page) to tell the user what series or episode to go listen to.

---

## Natural language structure

Episode: 005 Drinking your own Champagne
Link to episode: http://…
Tags: champagne, software development, product discovery, product ownership
Series: New Product Development
Link to series: http://…
Description:
Bob Dunderhauson and Pill Hear are talkin today...
Transcript:
We need to create a field of champagne grapes and
….

Episode: 006 Taking out the Trash
Link to episode: http://…
Tags: ….
Series: New Product Development
Link to series: http://…
Transcript:

---

## Columnar structure

Given that the goal is to direct people to podcast content, the first column will be how you care to reference podcasts in spoken language so the user can take it as a lead or a link which the user can click on (either way it’s something actionable by the user).

Different podcasts have different needs. Some podcasts like LeadingAgile SoundNotes for example, has highly episodic content where you can listen to the 89th episode and not even care or be aware of the 88th episode or the 90th episode. Other podcasts like Agile Thoughts have series made up of many episodes so the content is serial. Table 1 shows a data structure for episodic podcasts. The first column is “human friendly reference” that people want from the concierge. The other columns are the features used to decide “what’s a good match” for the user, and to be more helpful, along with other information the user will want to know (such as a link to the series). Sometimes an episodic podcast will run a series so there is a feature for that. Table 2 shows the data structure for a podcast that runs series, in which case the concierge will be recommending the series since for serialized content, a user will want to listen to the entire series just as a home buyer buys a house rather than only a room.

**Table 1: data structure for episodic podcasts**

| Episode name | Tags | Summary (or transcript) | Series name | Link to episode | Link to series |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

**Table 2: data structure for serialized podcasts**

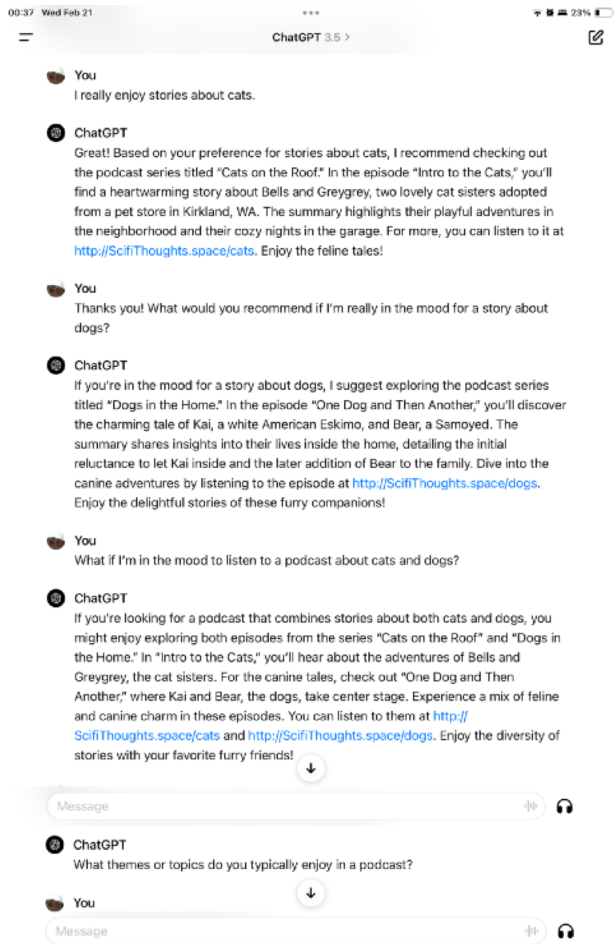
| Series name | Episode name | Tags | Summary (or transcript) | Link to series page |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

**Hypotheses to test: having the data in a columnar format is better for generation than a natural language structure.**
We found this hypotheses to be false. It didn't make a material difference on columnar format versus documents. We found that having a document per episode would work best as the logic is more simple than having a CSV with variable field sizes.

**Hypotheses to test: due to the data structure, the LLM will be able to use features (such as series or episodic links) to provide good actionable advice to the user.**
We found this hypothesis it be true. As shown in the next two figures, the LLM was able to identify CSV data and be coached to respond with helpful advice using that data.

**What about chunking?** The problem with columnar structure is that chunks can't be optimized for non-breaking between episodes. If instead of adding rows to a table, a separate document is made for each podcast, and then it's likely that one podcast worth of content will fit into one chunk. This also allows the use of metadata per document to store more ephemeral things like links and tags.

Getting advice from concierge




---

## How much data to put in transcript

Regarding the summary data, using the entire transcript will increase the semantic background of the content.  On the other hand, maybe a summary would be good enough and will provide less of a burden on data storage. The amount of effort to create a summary in the data pipeline is going to be the same amount whether the embedding contains the summary or the full transcript. For this project we are going with the full transcript and let the LLM decide how to generate a response for the user.

## 4. Feature Engineering

The link-columns (or headers marked with colons) help in generating a useful response so the user can simply click and go to the content.

## 5. Data Format

Experimentation and simplicity of maintenance lead to using natural language structure, with a document per episode.

# 6. Chunking

The impact of poor tokenization (more commonly called chunking in the ML community) is believed to have a large impact on the quality of response generation.  Since we decided on natural language structure, this has greatly simplified chunking.  Ideally, our longest podcast transcript will fit in one chunk so generation will always be able to find the opening information about the episode name, series name, and links.  The majority of podcasts are less than thirty minutes.

Since a lot of time was spent on using columnar structure, the below goes over the complexity of using columnar structure where one of the columns has an unfixed record size.

---

## CSV

For CSV data, it's best to fit a row of data alone within a single chunk. For different types of data (non-csv or not-record based) this answer may not apply. This answer can be generalized to all row based data, independent of CSV format.

Background: Since it is CSV data, it's implied that content within a row has a strong semantic relationship and that there is little to no semantic relationship with the next row or previous, ie, row ordering can be random because the rows are independent of each other.
So when generating embedding for this kind of data, where the LLM is to generate responses using semantic meaning between rows, the goal is that each row of the CSV becomes a vector so that when a LLM is queried, it generates answers oriented around the semantic content among various rows (which is the goal in this case), which means these answers are based upon the fit among the CSV.
For more background Chunking Strategies for LLM Applications is a good source.

The chunking size in table three will cause the concierge to often "hit" on the semantically rich data in summary, and episode name in different chunks and then miss its association with referential information in Series name and links. The record structure has been broken and it will take luck that the relationship among those parts can be recreated.

**Table 3: chunking too small**

| Episode name | Tags | Summary (or transcript) | Series name | Link to episode | Link to series |
|---|---|---|---|---|---|
| 001 Introducing SciFi Thoughts | Science fiction | SciFi Thoughts is a podcast that will … | | Http:// SciFiThoughts. space/… | |
| 002 Cyberpunk | science fiction, cyberpunk | Cyberpunk is a genre born out of… | | Http:// SciFiThoughts. space/… | |
| 003 Clarkesworld Magazine | Science fiction, magazine, short story | Editor Neil Clarke started Clarkesworld… | Clarkesworld Magazine | Http:// SciFiThoughts. space/… | Http:// SciFiThoughts. space/… |
| 004 What does editor Neil Clarke fear most? | Science fiction, magazine, editor | In today's episode we dive into what does Neil… | Clarkesworld Magazine | Http:// SciFiThoughts. space/… | Http:// SciFiThoughts. space/… |

Table 4 shows what chunking will look like if it's too large. The confer I've will have better quality with this since probabilities are better that the relationships will be preserved but there will be cases when answer quality will be low due to bad luck with chunking. If I ask the concierge to direct me to episodes about cyberpunk , I could get the episode link from the row above. (See yellow chunk.)

**Table 4: chunking too large**

| Episode name | Tags | Summary (or transcript) | Series name | Link to episode | Link to series |
|---|---|---|---|---|---|
| 001 Introducing SciFi Thoughts | Science fiction | SciFi Thoughts is a podcast that will … | | Http:// SciFiThoughts. space/… | |
| 002 Cyberpunk | science fiction, cyberpunk | Cyberpunk is a genre born out of… | | Http:// SciFiThoughts. space/… | |
| 003 Clarkesworld Magazine | Science fiction, magazine, short story | Editor Neil Clarke started Clarkesworld… | Clarkesworld Magazine | Http:// SciFiThoughts. space/… | Http:// SciFiThoughts. space/… |
| 004 What does editor Neil Clarke fear most? | Science fiction, magazine, editor | In today's episode we dive into what does Neil… | Clarkesworld Magazine | Http:// SciFiThoughts. space/… | Http:// SciFiThoughts. space/… |

**Hypothesis to test: the best quality advice will happen when the chunks match the row size.**

If the above hypothesis is true, then a max row size must be established so a row isn't fragmented by chunking.

# 7. Embedding Generation

# 8. Normalization

# 9. Storage

Each embedding will be a row in a Vector database and our embeddings are dimensioned at 1536. This implies that for a weekly podcast, there will be 52 embeddings or data rows added to a vector database per year, 520 rows after ten years (520 rows X 1536 length X 32 bits per element is approximately 6MB per decade.

This should be sustainable in performance for well over a hundred years.

# Updating and Sustaining  Concierge Knowledge
The podcast domain has the advantage that only new content is produced and that historical information is static.  This means the previous vectors in the DB stay relevant and don't need to be changed.
Since RSS feeds are updated upon podcast publication, a watcher pulls hourly for new content (is it possible to do push with RSS?) which starts the data pipeline and results in a new fresh embedding.

# Architecture as implemented
Data pipeline: RSS feed input -> transcript created -> data file caretaker -> tokenization -> Embedding -> Storage
Inference: OpenAPI, Vector DB, agent and orchestrator using Flowise.ai.

# Application to the Market
Due to the low performance of chatbots in the past, simply attaching the chatbot to a busy home page isn't going to attract much  attention. The approach we plan to take is to popup the Concierge when a user visits the podcast's website and ask them a compelling question to start interaction. The popup agent can be manually found with an attractive UI element and we propose standardizing on a URL for agents such as: help.leadingagile.com or assistant.leadingagile.com.

Since 40% of the traffic comes in from "the side door," (via the podcast player). we know we don't want to invest in ONLY improving a single page on a podcast client's website.

## Other Utility

The chatbot offers to email resources to user to encourage further traffic and build relationships.

# Behaviors for Agent Interaction
Good chat bot versus bad:
Don't popup and say "tell me what your think about"
Instead popup and tell them something valuable at the beginning.

The below scenarios are for one of the clients of the podcast recommender system, SciFi Thoughts.

Scenario: casual user meets agent
Given user on a webpage
When agent triggered
Then the agent invites the user to interact.

Scenario: user on a search mission
Given user interacts with agent
When user asks vague questions or asks “what is” SciFi Thoughts
Then tell them summary of podcast mission and direct to episode 1, and ask user a relationship building question.

Scenario: user asking weird questions

Given user is not knowledgeable
When they ask an unknowable question
Then tell them what you’re good at and ask them a relationship building question related to SciFi.

Scenario: Agent asks user a relationship building question
Background: stay in this scenario until the user either leaves, provides content that the agent can use for recommendation, or hits the patience limit and the agent closes the conversation.
Given Agent wants to build relationship with user
When Agent asks user a relationship building question related to science fiction
Then user responds with semantic meaning that the agent then uses to provide a suggestion.
Examples:
“What’s your favorite science fiction author, book, video, or movie?”
“When did you start enjoying science fiction?”
“Are you interested in science? What science are you most curious about?”
“Do you know what Space Opera is?”
“Have you heard of Trump Punk?”
“Do you like science fiction magazines?”
“Have you read an anthology?” “Can I tell you what an anthology is?”
“What do you think science fiction is?”
“Do you know when the first science fiction novel was written?”
“Do you have young children who would be interested in young adult science fiction?”

Scenario: veteran user
Given user wants to find something specific
When they respond with what they are looking for
Then respond with what series or episodes they will be interested in.

Scenario: Agent hits patience limit
Given the user seems to not be interested in anything other than chatting
When the Agent is on it's Nth interaction and N is greater than <patience limit>
Then tell user, "I've got to go. I'm reading a Lancer Kind story."

# How to measure success.

Just like when web search came out, it will take time to build the habit into the internet community of using agents for help. Up to this time, most agents are not helpful and useless so it will take some promoting that there is such an agent with advanced capability, forcing users into some interactions with the agent, and designing the agent interface so it catches the interest of users. Here are some metrics using the Goal, Question, Metric format.

Goal: users get value using the podcast concierge

- Question: has anyone noticed the concierge?
  - Metric: number of people who interacted with the concierge once
- Question: has anyone enjoyed interacting with the concierge?
  - Metric: Poll user for feedback on the experience. Count negatives and positives.
  - Metric: count number of times a user interacted with the agent more than once in a session.
  - Metric: count the number of times a user clicked on a link provided by the agent

If no one uses the agent, then the promotion strategy failed. If no one ever clicks on a link provided by the agent, then the user likely didn't care about the information provided to them.

# Citations

Setup a Flowise.ai document chat agent
CVS and Flowise