Building search-based RAG using Claude, Datasette and Val Town

21st June 2024

Retrieval Augmented Generation (RAG) is a technique for adding extra “knowledge” to systems built on LLMs, allowing them to answer questions against custom information not included in their training data. A common way to implement this is to take a question from a user, translate that into a set of search queries, run those against a search engine and then feed the results back into the LLM to generate an answer.

I built a basic version of this pattern against the brand new Claude 3.5 Sonnet language model, using SQLite full-text search running in Datasette as the search backend and Val Town as the prototyping platform.

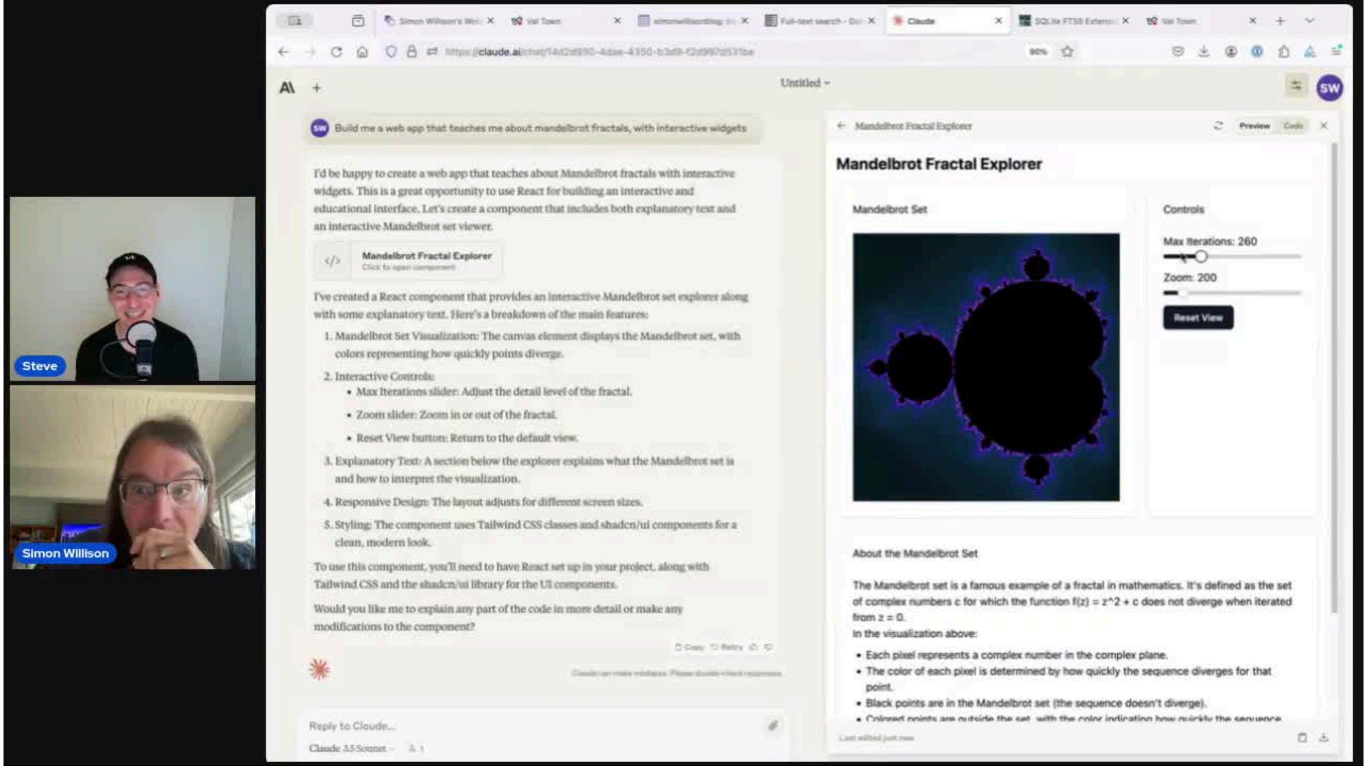
The implementation took just over an hour, during a live coding session with Val.Town founder Steve Krouse. I was the latest guest on Steve’s live streaming series where he invites people to hack on projects with his help.

You can watch the video below or on YouTube. Here are my own detailed notes to accompany the session.

Val Town Livestream with Simon Willison

Bonus: Claude 3.5 Sonnet artifacts demo

We started the stream by chatting a bit about the new Claude 3.5 Sonnet release. This turned into an unplanned demo of their “artifacts” feature where Claude can now build you an interactive web page on-demand.



Build me a web app that teaches me about mandelbrot fractals, with interactive widgets

I'd be happy to create a web app that teaches about Mandelbrot fractals with interactive widgets. This is a great opportunity to use React for building an interactive and educational interface. Let's create a component that includes both explanatory text and an interactive Mandelbrot set viewer.

I've created a React component that provides an interactive Mandelbrot set explorer along with some explanatory text. Here's a breakdown of the main features:

1. Mandelbrot Set Visualization: The canvas element displays the Mandelbrot set, with colors representing how quickly points diverge.
2. Interactive Controls:
   - Max Iterations slider: Adjust the detail level of the fractal.
   - Zoom slider: Zoom in or out of the fractal.
   - Reset View button: Return to the default view.
3. Explanatory Text: A section below the explorer explains what the Mandelbrot set is and how to interpret the visualization.
4. Responsive Design: The layout adjusts for different screen sizes.
5. Styling: The component uses Tailwind CSS classes and shadcn/ui components for a clean, modern look.

To use this component, you'll need to have React set up in your project, along with Tailwind CSS and the shadcn/ui library for the UI components.

Would you like me to explain any part of the code in more detail or make any modifications to the component?

About the Mandelbrot Set

The Mandelbrot set is a famous example of a fractal in mathematics. It's defined as the set of complex numbers c for which the function f(z) = z^2 + c does not diverge when iterated from z = 0.

In the visualization above:

- Each pixel represents a complex number in the complex plane.
- The color of each pixel is determined by how quickly the sequence diverges for that point.
- Black points are in the Mandelbrot set (the sequence doesn't diverge).
- Colored points are outside the set, with the color indicating how quickly the sequence

At 3m02s I prompted it with:

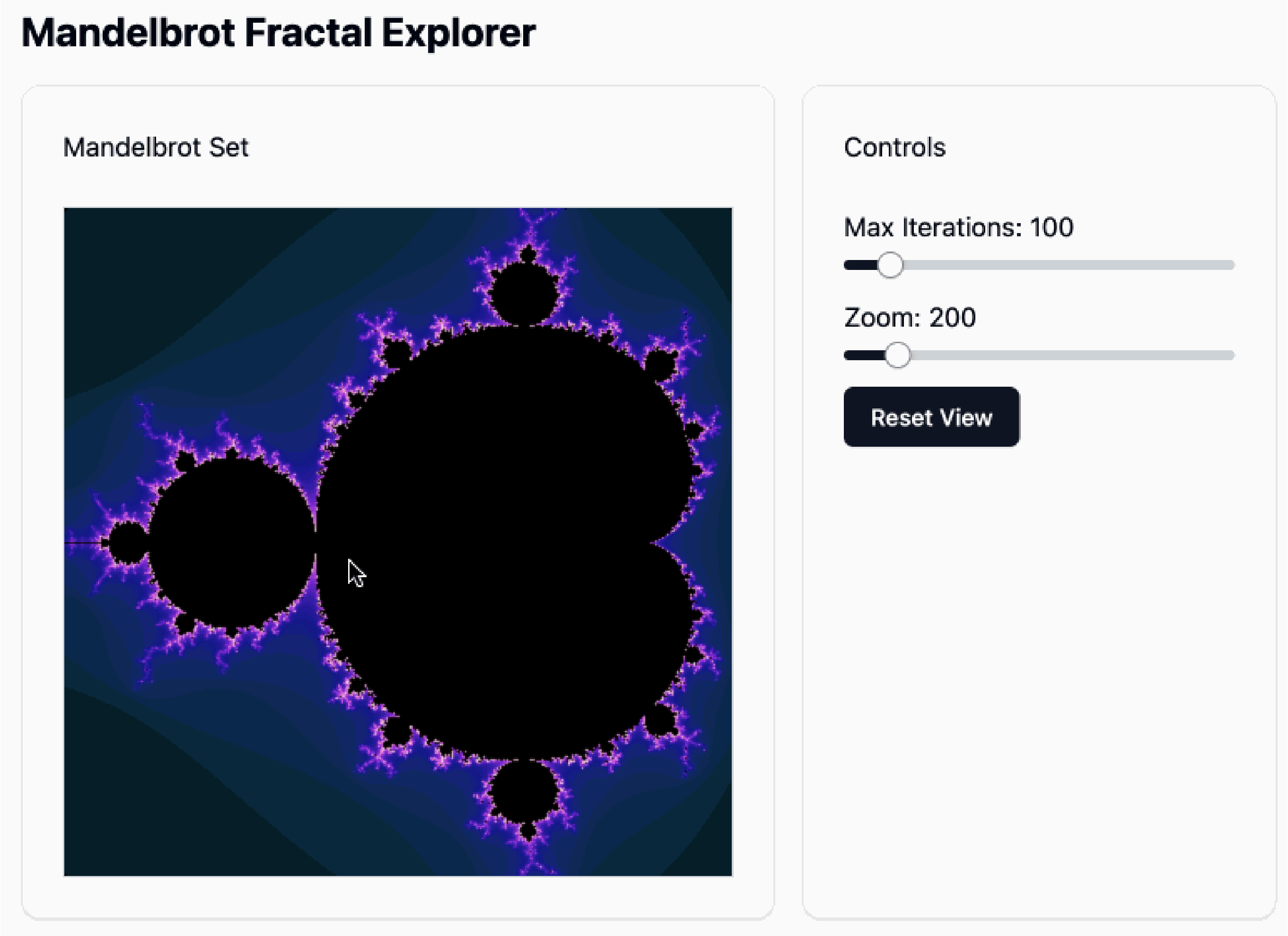
> Build me a web app that teaches me about mandelbrot fractals, with interactive widgets

This worked! Here's the code it wrote—I haven't yet found a good path for turning that into a self-hosted interactive page yet.

This didn't support panning, so I added:

> Again but let me drag on the canvas element to pan around

Which gave me this. Pretty impressive!



Max Iterations: 100

Zoom: 200

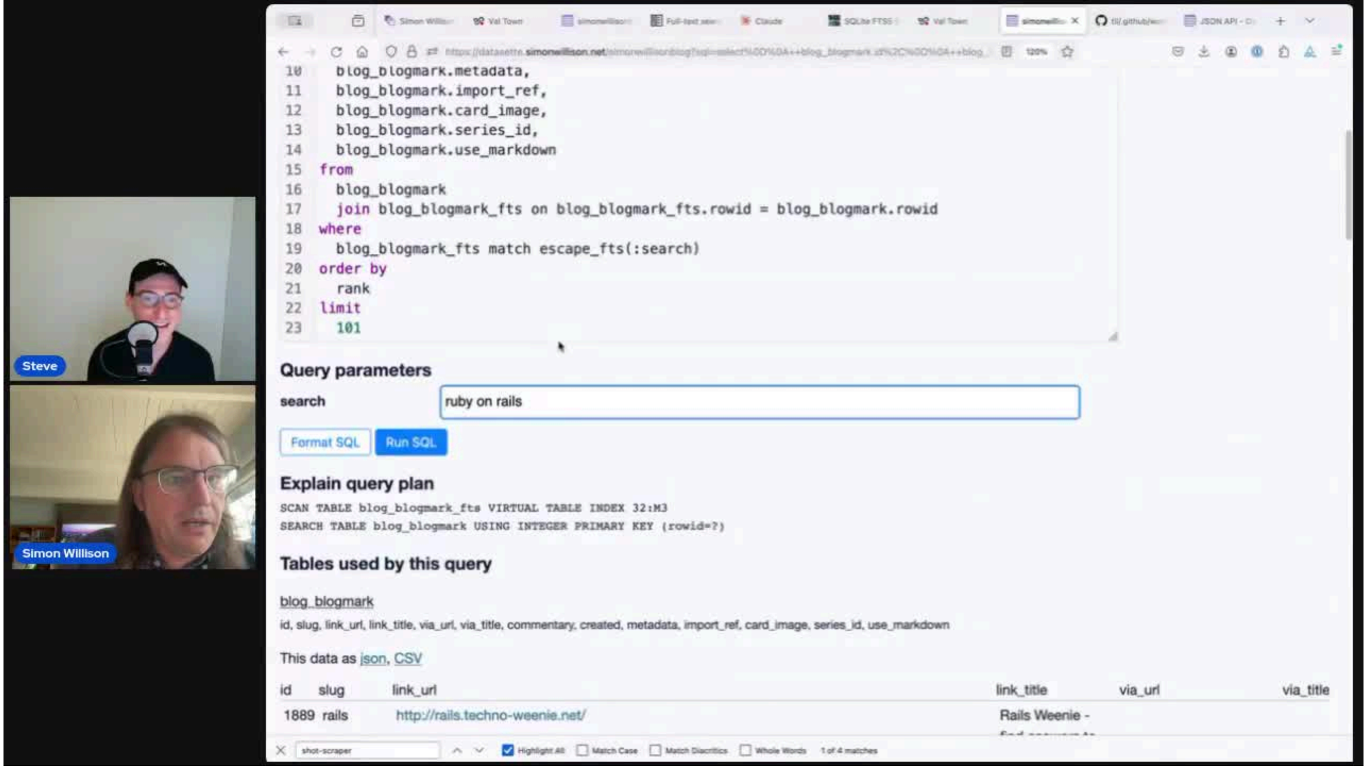
Ingredients for the RAG project

RAG is often implemented using [vector search against embeddings](), but there's an alternative approach where you turn the user's question into some full-text search queries, run those against a traditional search engine, then feed the results back into an LLM and ask it to use them to answer the question.

SQLite includes [surprisingly good full-text search](), and I've built a lot of tools against that in the past—including [sqlite-utils enable-fts]() and [Datasette's FTS features]().

My blog has a lot of content, which lives in a Django PostgreSQL database. But I also have a GitHub Actions repository which [backs up that data]() as JSON, and then publishes a SQLite copy of it to [datasette.simonwillison.net]()—which means I have a Datasette-powered JSON API for running searches against my content.

Let's use that API to build a question answering RAG system!



```
  blog_blogmark.metadata,
  blog_blogmark.import_ref,
  blog_blogmark.card_image,
  blog_blogmark.series_id,
  blog_blogmark.use_markdown
from
  blog_blogmark
  join blog_blogmark_fts on blog_blogmark_fts.rowid = blog_blogmark.rowid
where
  blog_blogmark_fts match escape_fts(:search)
order by
  rank
limit
  101
```

Query parameters

search: ruby on rails

Explain query plan

```
SCAN TABLE blog_blogmark_fts VIRTUAL TABLE INDEX 32:M3
SEARCH TABLE blog_blogmark USING INTEGER PRIMARY KEY (rowid=?)
```

Tables used by this query

blog_blogmark

id, slug, link_url, link_title, via_url, via_title, commentary, created, metadata, import_ref, card_image, series_id, use_markdown

This data as json, CSV

id	slug	link_url	link_title	via_url	via_title
1889	rails	http://rails.techno-weenie.net/	Rails Weenie -		

Step one then was to prototype up a SQL query we could use with that API to get back search results. After some iteration I got to this:

```sql
select
  blog_entry.id,
  blog_entry.title,
  blog_entry.body,
  blog_entry.created
from
  blog_entry
  join blog_entry_fts on blog_entry_fts.rowid = blog_entry.rowid
where
  blog_entry_fts match :search
order by
  rank
limit
  10
```

Try that here. The query works by joining the `blog_entry` table to the SQLite FTS `blog_entry_fts` virtual table, matched against the `?search=` parameter from the URL.

When you join against a FTS table like this a `rank` column is exposed with the relevance score for each match.

Adding `.json` to the above URL turns it into an API call... so now we have a search API we can call from other code.

A plan for the build

We spent the rest of the session writing code in Val Town, which offers a browser editor for a server-side Deno-based environment for executing JavaScript (and TypeScript) code.

The finished code does the following:

1. Accepts a user’s question from the `?question=` query string.
2. Asks Claude 3.5 Sonnet to turn that question into multiple single-word search queries, using a Claude function call to enforce a schema of a JSON list of strings.
3. Turns that list of keywords into a SQLite FTS query that looks like this: `"shot-scraper" OR "screenshot" OR "web" OR "tool" OR "automation" OR "CLI"`
4. Runs that query against Datasette to get back the top 10 results.
5. Combines the title and body from each of those results into a longer context.
6. Calls Claude 3 again (originally Haiku, but then we upgraded to 3.5 Sonnet towards the end) with that context and ask it to answer the question.
7. Return the results to the user.

The annotated final script

Here’s the final script we ended up with, with inline commentary. Here’s the initial setup:

```
import Anthropic from "npm:@anthropic-ai/sdk@0.24.0";

/* This automatically picks up the API key from the ANTHROPIC_API_KEY
environment variable, which we configured in the Val Town settings */
const anthropic = new Anthropic();
```

We’re using the very latest release of the Anthropic TypeScript SDK, which came out just a few hours prior to recording the livestream.

I set the `ANTHROPIC_API_KEY` environment variable to my Claude 3 API key in the Val Town settings, making it available to all of my Vals. The `Anthropic()` constructor picks that up automatically.

Next, the function to suggest keywords for a user’s question:

```
async function suggestKeywords(question) {
  // Takes a question like "What is shot-scraper?" and asks 3.5 Sonnet
  // to suggest individual search keywords to help answer the question.
  const message = await anthropic.messages.create({
    max_tokens: 128,
    model: "claude-3-5-sonnet-20240620",
    // The tools option enforces a JSON schema array of strings
    tools: [{
      name: "suggested_search_keywords",
      description: "Suggest individual search keywords to help answer the question.",
      input_schema: {
        type: "object",
        properties: {
          keywords: {
            type: "array",
            items: {
              type: "string",
            },
            description: "List of suggested single word search keywords",
          },
        },
        required: ["keywords"],
      },
```

```
    }],
    // This forces it to always run the suggested_search_keywords tool
    tool_choice: { type: "tool", name: "suggested_search_keywords" },
    messages: [
      { role: "user", content: question },
    ],
  });
  // This helped TypeScript complain less about accessing .input.keywords
  // since it knows this object can be one of two different types
  if (message.content[0].type == "text") {
    throw new Error(message.content[0].text);
  }
  return message.content[0].input.keywords;
}
```

We’re asking Claude 3.5 Sonnet here to suggest individual search keywords to help answer that question. I tried Claude 3 Haiku first but it didn’t reliably return single word keywords—Sonnet 3.5 followed the “single word search keywords” instruction better.

This function also uses Claude tools to enforce a response in a JSON schema that specifies an array of strings. More on how I wrote that code (with Claude’s assistance) later on.

Next, the code to run the search itself against Datasette:

```
// The SQL query from earlier
const sql = `select
  blog_entry.id,
  blog_entry.title,
  blog_entry.body,
  blog_entry.created
from
  blog_entry
  join blog_entry_fts on blog_entry_fts.rowid = blog_entry.rowid
where
  blog_entry_fts match :search
order by
  rank
limit
  10`;

async function runSearch(keywords) {
  // Turn the keywords into "word1" OR "word2" OR "word3"
  const search = keywords.map(s => `"${s}"`).join(" OR ");
  // Compose the JSON API URL to run the query
  const params = new URLSearchParams({
    search,
    sql,
    _shape: "array",
  });
  const url = "https://datasette.simonwillison.net/simonwillisonblog.json?" + params;
  const result = await (await fetch(url)).json();
  return result;
}
```

Datasette supports read-only SQL queries via its JSON API, which means we can construct the SQL query as a JavaScript string and then encode it as a query string using `URLSearchParams()`.

We also take the list of keywords and turn them into a SQLite FTS search query that looks like `"word1" OR "word2" OR "word3"`.

SQLite's built-in relevance calculations work well with this—you can throw in dozens of words separated by `OR` and the top ranking results will generally be the ones with the most matches.

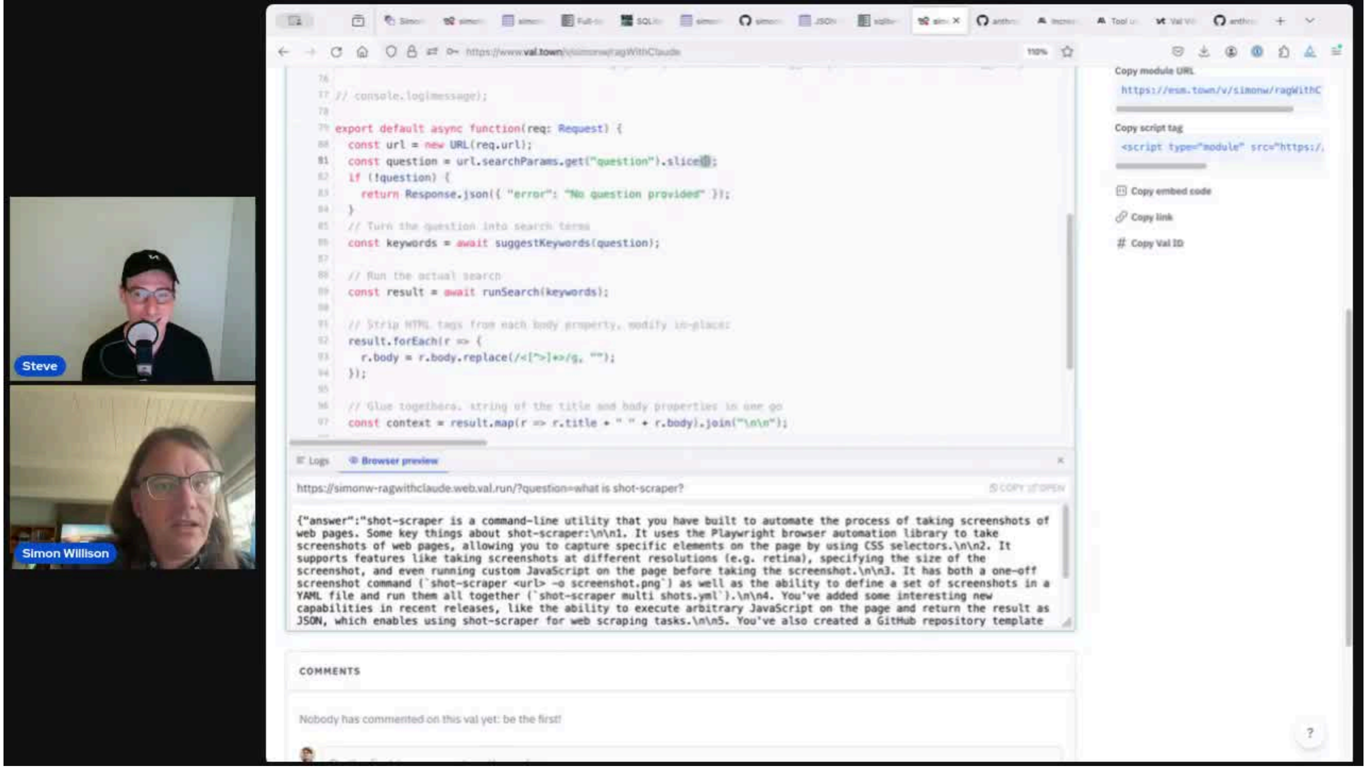
Finally, the code that ties this together—suggests keywords, runs the search and then asks Claude to answer the question. I ended up bundling that together in the HTTP handler for the Val Town script—this is the code that is called for every incoming HTTP request:

```
export default async function(req: Request) {
  // This is the Val Town HTTP handler
  const url = new URL(req.url);
  const question = url.searchParams.get("question").slice(0, 40);
  if (!question) {
    return Response.json({ "error": "No question provided" });
  }
  // Turn the question into search terms
  const keywords = await suggestKeywords(question);

  // Run the actual search
  const result = await runSearch(keywords);

  // Strip HTML tags from each body property, modify in-place:
  result.forEach(r => {
    r.body = r.body.replace(/<[^>]*>/g, "");
  });

  // Glue together a string of the title and body properties in one go
  const context = result.map(r => r.title + " " + r.body).join("\n\n");

  // Ask Claude to answer the question
  const message = await anthropic.messages.create({
    max_tokens: 1024,
    model: "claude-3-haiku-20240307",
    messages: [
      { role: "user", content: context },
      { role: "assistant", content: "Thank you for the context, I am ready to answer your question" },
      { role: "user", content: question },
    ],
  });
  return Response.json({answer: message.content[0].text});
}
```

There are many other ways you could arrange the prompting here. I quite enjoy throwing together a fake conversation like this that feeds in the context and then hints at the agent that it should respond next with its answer, but there are many potential variations on this theme.

This initial version returned the answer as a JSON object, something like this:

```
{
    "answer": "shot-scraper is a command-line tool that automates the process of taking screenshots of web pages..."
}
```



```
// console.log(message);

export default async function(req: Request) {
  const url = new URL(req.url);
  const question = url.searchParams.get("question").slice();
  if (!question) {
    return Response.json({ "error": "No question provided" });
  }
  // Turn the question into search terms
  const keywords = await suggestKeywords(question);

  // Run the actual search
  const result = await runSearch(keywords);

  // Strip HTML tags from each body property, modify in-place:
  result.forEach(r => {
    r.body = r.body.replace(/<[^>]+>/g, "");
  });

  // Glue together a string of the title and body properties in one go
  const context = result.map(r => r.title + " " + r.body).join("\n\n");
```

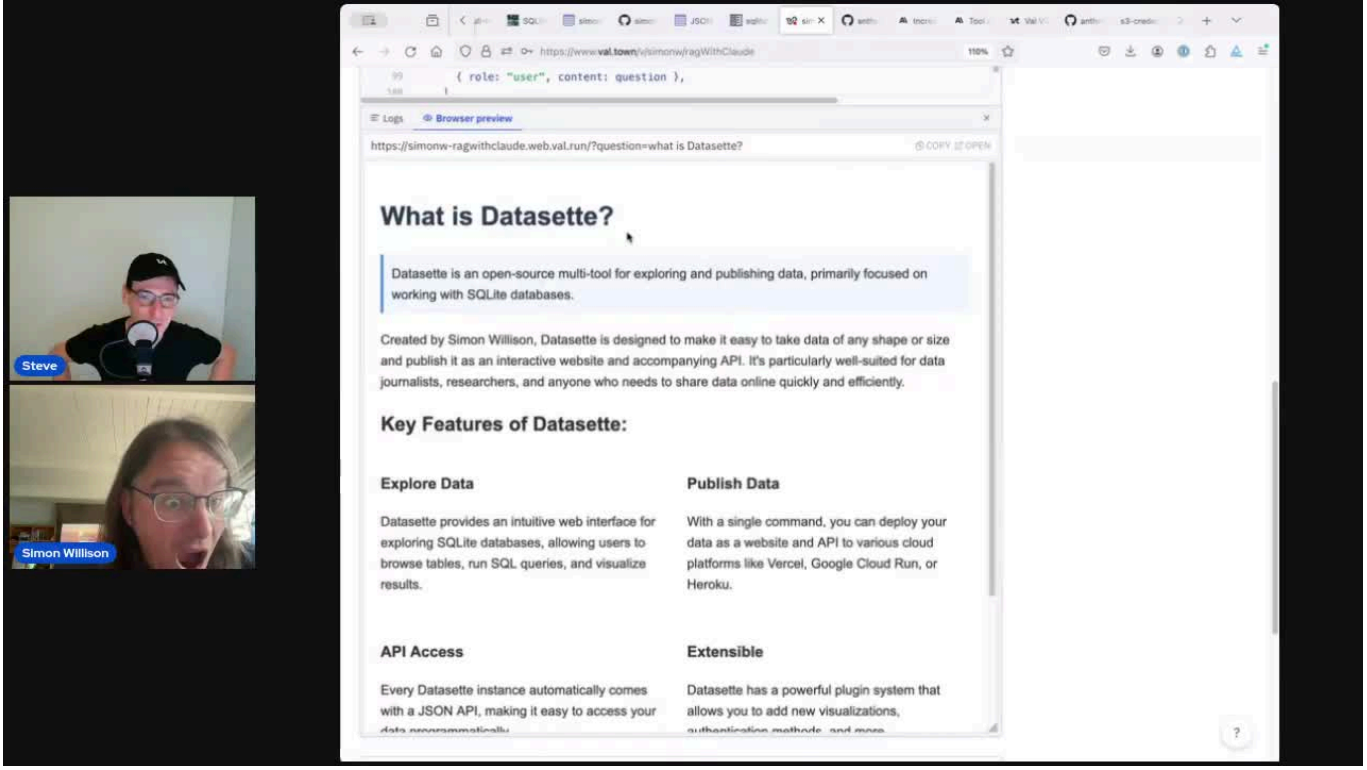
https://simonw-ragwithclaude.web.val.run/?question=what is shot-scraper?

```
{"answer":"shot-scraper is a command-line utility that you have built to automate the process of taking screenshots of
web pages. Some key things about shot-scraper:\n\n1. It uses the Playwright browser automation library to take
screenshots of web pages, allowing you to capture specific elements on the page by using CSS selectors.\n\n2. It
supports features like taking screenshots at different resolutions (e.g. retina), specifying the size of the
screenshot, and even running custom JavaScript on the page before taking the screenshot.\n\n3. It has both a one-off
screenshot command (`shot-scraper <url> -o screenshot.png`) as well as the ability to define a set of screenshots in a
YAML file and run them all together (`shot-scraper multi shots.yml`).\n\n4. You've added some interesting new
capabilities in recent releases, like the ability to execute arbitrary JavaScript on the page and return the result as
JSON, which enables using shot-scraper for web scraping tasks.\n\n5. You've also created a GitHub repository template
```

We were running out of time, but we wanted to add an HTML interface. Steve suggested getting Claude to write the whole thing! So we tried this:

```
  const message = await anthropic.messages.create({
    max_tokens: 1024,
    model: "claude-3-5-sonnet-20240620", // "claude-3-haiku-20240307",
    system: "Return a full HTML document as your answer, no markdown, make it pretty
with exciting relevant CSS",
    messages: [
      { role: "user", content: context },
      { role: "assistant", content: "Thank you for the context, I am ready to answer
your question as HTML" },
      { role: "user", content: question },
    ],
  });
  // Return back whatever HTML Claude gave us
  return new Response(message.content[0].text, {
    status: 200,
    headers: { "Content-Type": "text/html" }
  });
```

We upgraded to 3.5 Sonnet to see if it had better “taste” than Haiku, and the results were really impressive. Here’s what it gave us for “What is Datasette?”:



```
{ role: "user", content: question },
```

https://simonw-ragwithclaude.web.val.run/?question=what is Datasette?

What is Datasette?

Datasette is an open-source multi-tool for exploring and publishing data, primarily focused on working with SQLite databases.

Created by Simon Willison, Datasette is designed to make it easy to take data of any shape or size and publish it as an interactive website and accompanying API. It's particularly well-suited for data journalists, researchers, and anyone who needs to share data online quickly and efficiently.

Key Features of Datasette:

Explore Data

Datasette provides an intuitive web interface for exploring SQLite databases, allowing users to browse tables, run SQL queries, and visualize results.

Publish Data

With a single command, you can deploy your data as a website and API to various cloud platforms like Vercel, Google Cloud Run, or Heroku.

API Access

Every Datasette instance automatically comes with a JSON API, making it easy to access your data programmatically.

Extensible

Datasette has a powerful plugin system that allows you to add new visualizations, authentication methods, and more.

It even styled the page with flexbox to arrange the key features of Datasette in a 2x2 grid! You can see that in the video at 1h13m17s.

There’s a full copy of the final TypeScript code available in a Gist.

Some tricks we used along the way

I didn’t write all of the above code. Some bits of it were written by pasting things into Claude 3.5 Sonnet, and others used the Codeium integration in the Val Town editor (described here).

One pattern that worked particularly well was getting Sonnet to write the tool-using TypeScript code for us.

The Claude 3 documentation showed how to do that using curl. I pasted that `curl` example in, added some example TypeScript and then prompted:

> Guess the JavaScript for setting up a tool which just returns a list of strings, called suggested_search_keywords

Here’s my full prompt:

```
#!/bin/bash
IMAGE_URL="https://upload.wikimedia.org/wikipedia/commons/a/a7/Camponotus_flavomargin
IMAGE_MEDIA_TYPE="image/jpeg"
IMAGE_BASE64=$(curl "$IMAGE_URL" | base64)
curl https://api.anthropic.com/v1/messages \
     --header "content-type: application/json" \
     --header "x-api-key: $ANTHROPIC_API_KEY" \
     --header "anthropic-version: 2023-06-01" \
     --data \
'{
    "model": "claude-3-sonnet-20240229",
    "max_tokens": 1024,
```

```
    "tools": [{
        "name": "record_summary",
        "description": "Record summary of an image using well-structured JSON.",
        "input_schema": {
            "type": "object",
            "properties": {
                "key_colors": {
                    "type": "array",
                    "items": {
                        "type": "object",
                        "properties": {
                            "r": { "type": "number", "description": "red value [0.0,
1.0]" },
                            "g": { "type": "number", "description": "green value
[0.0, 1.0]" },
                            "b": { "type": "number", "description": "blue value [0.0,
1.0]" },
                            "name": { "type": "string", "description": "Human-
readable color name in snake_case, e.g. \"olive_green\" or \"turquoise\"" }
                        },
                        "required": [ "r", "g", "b", "name" ]
                    },
                    "description": "Key colors in the image. Limit to less then
four."
                },
                "description": {
                    "type": "string",
                    "description": "Image description. One to two sentences max."
                },
                "estimated_year": {
                    "type": "integer",
                    "description": "Estimated year that the images was taken, if is
it a photo. Only set this if the image appears to be non-fictional. Rough estimates
are okay!"
                }
            },
            "required": [ "key_colors", "description" ]
        }
    }],
    "tool_choice": {"type": "tool", "name": "record_summary"},
    "messages": [
        {"role": "user", "content": [
            {"type": "image", "source": {
                "type": "base64",
                "media_type": "'$IMAGE_MEDIA_TYPE'",
                "data": "'$IMAGE_BASE64'"
            }},
            {"type": "text", "text": "Describe this image."}
        ]}
    ]
}'
```

Based on that example and this JavaScript code:

```
const anthropic = new Anthropic();
const message = await anthropic.messages.create({
```

```
  max_tokens: 1024,
  system: "Suggest individual search keywords to help answer this question. No
yapping.",
  messages: [
    { role: "user", content: question },
  ],
  model: "claude-3-haiku-20240307",
});
console.log(message.content[0].text);

Guess the JavaScript for setting up a tool which just returns a list of strings,
called suggested_search_keywords
```

It gave me back this, which was the *exact* code I needed to make my tool-enabled API call from Val Town:

```
const anthropic = new Anthropic();
const message = await anthropic.messages.create({
  max_tokens: 1024,
  model: "claude-3-haiku-20240307",
  tools: [{
    name: "suggested_search_keywords",
    description: "Suggest individual search keywords to help answer the question.",
    input_schema: {
      type: "object",
      properties: {
        keywords: {
          type: "array",
          items: {
            type: "string"
          },
          description: "List of suggested search keywords"
        }
      },
      required: ["keywords"]
    }
  }],
  tool_choice: { type: "tool", name: "suggested_search_keywords" },
  messages: [
    { role: "user", content: question }
  ]
});

console.log(message.content[0].text);
```

I really like this trick, and I use it often in my own work. Find some example code that illustrates the pattern you need, paste in some additional context and get the LLM to figure the rest out for you.

This is just a prototype

It’s important to reiterate that this is just a prototype—it’s the version of search-backed RAG I could get working in an hour.

Putting something like this into production requires a whole lot more work. Most importantly, good RAG systems are backed by evals—it's extremely hard to iterate on and improve a system like this if you don't have a good mechanism in place to evaluate if your changes are making things better or not. Your AI Product Needs Evals by Hamel Husain remains my favourite piece of writing on how to go about putting these together.

Additional links from the livestream

Here are some of the other projects and links mentioned during our conversation:

- Datasette and its 150+ plugins.
- My original idea for a project was to use the Datasette Write API and run scheduled Vals to import data from various sources (my toots, tweets, posts etc) into a single searchable table.
- LLM—my command-line utility for working with different language models.
- shot-scraper for automating screenshots and scraping websites with JavaScript from the command-line—here's a recent demo where I scraped Google using shot-scraper and fed the results into LLM as a basic form of RAG.
- My current list of 277 projects with at least one release on GitHub.
- My TIL blog, which runs on a templated version of Datasette—here's the database and here's the GitHub Actions workflow that builds it using the Baked Data pattern.
- I have some previous experiments using embeddings with Datasette, including a table of embeddings (encoded like this) for my TIL blog which I use to power related items. That's described in this TIL: Storing and serving related documents with openai-to-sqlite and embeddings.

Posted 21st June 2024 at 8:44 pm · Follow me on Mastodon or Twitter or subscribe to my newsletter

More recent articles

- Notes on the new Claude analysis JavaScript code execution tool - 24th October 2024
- Initial explorations of Anthropic's new Computer Use capability - 22nd October 2024
- Everything I built with Claude Artifacts this week - 21st October 2024

Part of series **How I use LLMs and ChatGPT**

14. Running OCR against PDFs and images directly in your browser - March 30, 2024, 5:59 p.m.
15. Building files-to-prompt entirely using Claude 3 Opus - April 8, 2024, 8:40 p.m.
16. AI for Data Journalism: demonstrating what we can do with this stuff right now - April 17, 2024, 9:04 p.m.
17. **Building search-based RAG using Claude, Datasette and Val Town** - June 21, 2024, 8:44 p.m.
18. django-http-debug, a new Django app mostly written by Claude - Aug. 8, 2024, 3:26 p.m.
19. Building a tool showing how Gemini Pro can return bounding boxes for objects in images - Aug. 26, 2024, 4:55 a.m.
20. Notes on using LLMs for code - Sept. 20, 2024, 3:10 a.m.

… more

projects 428 ai 859 datasette 415 prompt-engineering 97 generative-ai 725 llms 717

ai-assisted-programming 71 anthropic 86 claude 91 annotated-talks 23 val-town 5 rag 24

claude-artifacts 16 claude-3-5-sonnet 31

Next: Open challenges for AI engineering

Previous: Weeknotes: Datasette Studio and a whole lot of blogging