Level Up Coding

Photo by Dan Dimmock on Unsplash

I’m automating my Job as a Google Engineer with a custom-built Deep Research System - Here’s how to build it

Let’s dive into Deep Research, the most versatile next-gen RAG architecture out there. Let’s learn the pattern and build our own with Genkit.


Jakob Pörschmann Follow 10 min read · 4 days ago

“RAG is dead in 2025. It’s been solved, done over and over. It’s just nothing new anymore.” That’s what I heard from a fellow ML Engineer recently. Indeed, when discussing GenAI use cases with clients in 2025, I often find that standard RAG wouldn’t do the job anymore. The user queries they need to answer are usually way too complex and unspecific for RAG to handle them. They are also unable to train their users to ask specific enough questions that simple retrieval identifies the right context. The number and types of data sources are too complex, standardization turns out to be difficult.

Enter Deep Research. Deep Research is the most versatile and valuable Gen AI application design pattern out there right now. Every AI Engineer and leader in the space should understand Deep Research and how to customize it for their use case. And no, I’m not talking about the out-of-the-box deep researcher offered by Gemini, ChatGPT, etc., but your own custom implementation solving your domain-specific knowledge tasks.

In this article, we will discuss how to implement and generalize the Deep Research design pattern. The ultimate goal is then to specialize it to your domain use case. The full code repo is available here.

A Deep Researcher to answer technical questions about GCP

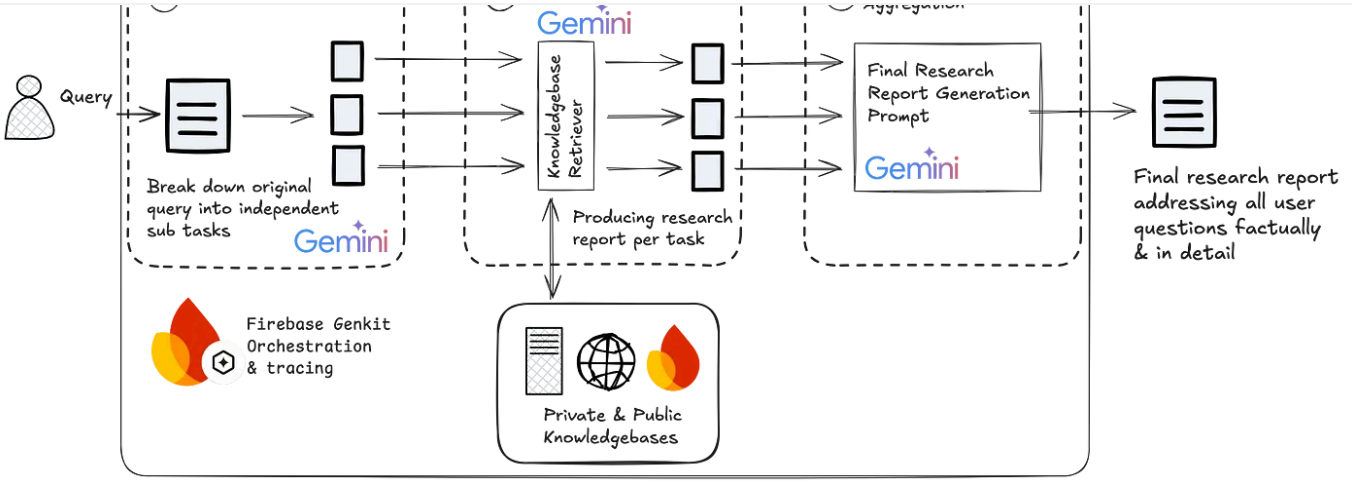
To have an example use case to work on, our aim is automating large parts of my job as Customer Engineer. After meeting clients I often spend time writing technical follow up emails providing answers to open questions and

resources such as documentation and sample code. Depending on the complexity this is a significant time invest, especially when done multiple times a week. We will build a Deep Research System to automate these technical follow-up emails. As an Orchestration framework, we will use Firebase Genkit.

Conceptual Overview

The deep Research pattern consists of three steps:

1. Understanding and breaking down the user request
2. Parallel Research & retrieval pipeline to gather information



Overview of the Deep Research Gen AI App Design Pattern

Let's build it!

Breaking down the user request

Our starting point is the initial meeting transcript. After the meeting, we need to extract the technical questions to follow up on. We should extract

these in a format that is easy to process afterwards.

This is easy to solve using a simple text prompt with the raw meeting transcript as input. Every task to extract is a plain string in question form. For example, a question could look as follows: "How do I build my own Deep Research Assistant with Gemini 2.5 and Genkit?" As we want our assistant to work on any meeting transcript, the number of tasks to extract should be flexible. Thus, the extraction prompt should output an array with an undefined number of strings.

Genkit in Typescript allows us to define input and output types for callable prompts. The input to the task extraction prompt is the raw transcript, thus a simple string. The output of the extraction prompt (array of research question objects) we define as follows:

```
const TaskSchema = z.object({
   description: z.string(),
});
```

```
const TaskArraySchema = ai.defineSchema(
   'TaskArraySchema',
   z.array(TaskSchema)
);
```

The extraction prompt should not be over-engineered, question extraction is a comparably simple task. The prompt should explain the context and rules of the extraction task. The LLM must not make up or leave out any questions present in the transcript. Furthermore, it should briefly summarize the questions while staying as factual as possible. To accomplish this we set the

temperature at the lower end. Using dotprompt, Genkit makes it easy to define a prompt together with prompt configuration, input, and output schema.

```
---
config:
  temperature: 0.1
input:
  schema:
    transcript: string
output:
  schema:
  TaskArraySchema
---

<< System Instructions >>
You are an AI assistant helping extract customer's technical questions from a co
Analyze the transcript and identify the core technical questions, needs, or pain

** Always include topics that the CE promised to send a follow up on.
** Always focus on the specific technical questions that can be answered based o
** Only focus on questions that have NOT YET been answered in the meeting.
** DO NOT include high level business and use case questions.

<< Output Formatting >>
Format the output as a JSON array of tasks, where each task has:

description: The customer's technical question, phrased as a concise question fr

<< Meeting Transcript to analyze >>

{{transcript}}

<< End of Meeting Transcript to analyze >>

<< Tasks (JSON array) >>
```

To orchestrate the extraction step, we define a Genkit flow defining a prompt callable as follows:

```
export const taskExtractionFlow = ai.defineFlow(
    {
        name: "taskExtractionFlow",
        inputSchema: z.string(),
        outputSchema: TaskArraySchema,
    },
    async (transcript) => {

        const taskExtractionPrompt = ai.prompt('taskExtraction');

        const { output } = await taskExtractionPrompt(
            {
                transcript: transcript,
            },
            {
                model: gemini25ProPreview0325,
                output: { schema: TaskArraySchema }
            }
        );

        return output;
    }
);
```

That's it, we are all set up for the task extraction. We finished the first step and can now dynamically extract the questions to research for each of our past meetings.

Parallel Task Research

Next, we need to research each of these questions. The complexity of the research step vastly differs with the size and type of knowledge base required to accomplish the research. Our application handles technical questions around GCP architecture decisions and tooling when building applications on GCP. We usually provide up-to-date information from the GCP documentation and sample code from the GCP GitHub repositories.

We choose two knowledge sources to provide the right context to the LLM:

First, assume we have custom-built knowledge with valuable GCP documentation chunks turned into text embeddings stored in Firestore. This is accessible to us via the Firestore Vector index functionality.

Second, we include the Google Programmable Search Engine API to simulate a Google search for every research task. The programmable search engine API allows us to tap into up-to-date documentation (indexed by Google). It also allows us to limit the search engine to a given set of websites and their respective subpages. That makes the programmable search engine API a convenient source of real-time information, without needing to re-build our knowledge base (whenever the documentation is updated).

Input for the task research is our previously created array of research tasks. The research result for each task should be typed as follows:

```
const TaskResearchResponse = ai.defineSchema(
   'TaskResearchResponse',
   z.object({
       answer: z.string(),
       caveats: z.array(z.string()),
       docReferences: z.array(z.object({
           title: z.string(),
           url: z.string(),
           relevantContent: z.string().optional(),
       })),
   })
);
```

This strictly typed research report allows us to carry over important caveats and relevant document references to the aggregation step. This structure is crucial to keep an oversight of which document resulted in which insight. It

allows us to provide documentation links together with the insights in our follow-up email.

Heart of the task research is the document retrieval. Genkit allows connecting knowledge bases via many pre-built plugins or as custom functions. In this case, we define two custom retrievers, one to run a vector similarity search on our Firestore Vector index, the other to run a search query via the Programmable Search Engine API.

The following is the definition of our Firestore Vector Search Genkit Retriever. It uses the vector search server-side action. This ssa is defined as a wrapper around the Firestore SDK (see the repo for the code). It then formats the nearest neighbouring documents from our knowledge base into an array of Genkit Documents.

```
export async function createSimpleFirestoreVSRetriever(ai: Genkit) {
    return ai.defineSimpleRetriever(
        {
            name: "simpleFirestoreVSRetriever",
            configSchema: z.object({
                limit: z.number().optional().default(5),
            }).optional(),
            // Specify how to get the main text content from the Document object
            content: (doc: Document) => doc.text,
            // Specify how to get metadata from the Document object
            metadata: (doc: Document) => ({ ...doc.metadata }), // Include all m
        },
        async (query, config) => {
            const results = await vectorSearchAction(query.text, { limit: config

            const resultDocs: Document[] = results.map(doc => {
                return Document.fromText(
                    doc.content || '', {
                    firestore_id: doc.documentId,
                    chunkId: doc.chunkId
                });
            });
```

```
            return resultDocs;
        }
    );
}
```

The custom search retriever calls our previously defined Programmable Search Engine. The Search Engine API only provides the result URLs. Thus, we parse the results and fetch the content from the top search results via the provided URL. We apply some basic cleaning to the raw HTML content and transform it into Genkit Documents for further processing. You can find the full code in the repo.

Next, we summarize the answer to the individual research question by populating the research question response schema using Gemini 2.5. The summarization prompt is general and simple:

```
---
config:
  temperature: 0.1
input:
  schema:
    task: string
    format_instructions: string
output:
  schema:
  TaskResearchResponse
---
Research the following technical task using the provided Google Cloud documentat

Task: {{task}}

Provide a response that:
1. Clearly answers the technical question
2. Includes specific steps or configurations where relevant
3. Notes any important caveats or best practices
4. References specific sections of the documentation
```

```
Technical Response:
```

Finally, we merge the individual task research steps into a Genkit flow. To process efficiently we wrap the retrieval steps into a promise per task and the research aggregation into an asynchronous map. That parallelizes the individual task research.

```
export const taskReseachFlow = ai.defineFlow(
    {
        name: "taskReseachFlow",
        inputSchema: TaskArraySchema,
        outputSchema: TaskResearchResponseArray,
    },
    async (tasks) => {

        console.log("Running Task Research Flow on transcript...");

        // 1. Retrieve relevant documents for each task in parallel
        const retrievalPromises = tasks.map(task => {
            return ai.retrieve({
                retriever: 'simpleFirestoreVSRetriever',
                query: task.description, // Use task description as query
                options: { limit: 10 }
            });
        });

        const taskDocsArray = await Promise.all(retrievalPromises);

        // 2. Parallel task research generation
        const taskResearchPrompt = ai.prompt('taskResearch');
        const generationPromises = tasks.map(async (task, index) => {
            const docs = taskDocsArray[index];
            const { output } = await taskResearchPrompt(
                {
                    question: task.description
                },
                {
                    docs: docs,
                    output: { schema: TaskResearchResponse }
                }
```

```
            );

            return output;
        });
        const researchResults = await Promise.all(generationPromises);

        return researchResults;
    }
);
```

Our flow returns an array of research result objects. The standardized format makes the result easiest to process in the next workflow step.

Aggregation of Research Results

We have come a long way already! We successfully extracted the tasks to research and built an efficient pipeline that processes any number of research tasks efficiently in parallel, tapping into multiple knowledge bases, public and private. All that's left is aggregating a report on our findings into a format, useful to us and our users. Our working example is the challenge of writing technical follow-up emails. Using the collection of individual research reports, let's formulate a text prompt that uses these reports to write a concise email draft that provides the answers and documentation resources.

The research report aggregation prompt will vastly differ from use case to use case. Out of the text prompts we have used so far, this one should be the most detailed as we need to define the desired output format, tone, etc. Since we are mostly adjusting for writing style, a multi-shot prompting approach makes sense here as well. With clean and diverse examples, the model will have an easier time following your styling guidance.

To generate our technical follow-up emails, let's use the following prompt:

```
---
config:
  temperature: 1
input:
  schema:
    tasks: string
    research: string
output:
  schema:
    email: string
---
Generate a professional follow-up email to the customer based on the technical r

Original Task:
{{tasks}}

Research Findings:
{{research}}

Requirements for the email:
1. Start with a brief meeting reference and summary
2. Address each technical question briefly and consicely, most bullet points sho
3. Link to specific documentation sections whenever possible, but only as it mak
5. Maintain a professional but friendly tone
6. End with next steps or an offer for further clarification

Here is an example of the conciseness level of the email.
In your email please match the level of details provided in the example.
<< BEGINNING OF EXAMPLE EMAIL >>
See example in the full codebase
<< END OF EXAMPLE EMAIL >>

Generated Email:
```

We should finally wrap the email generation prompt into a respective flow (code in the repo).

That concludes the Deep Research pipeline.

The complete code for this Deep Research workflow is available on GitHub. Fork the repo, explore the implementation details, and adapt it to your

research intensive tasks.

Conclusion & Next Steps

The Deep Research design pattern is one of the most useful Gen AI application architectures. Deep Research is the next-gen evolution of the standard RAG system. I’m convinced that in 1–2 years, many of the day-to-day knowledge worker tasks will be accelerated or automated using a Deep Research application The initial idea of this project was to automate a big chunk of my job. That is already working shockingly well.

The practical usefulness of any Deep Researcher, will vastly depend on the relevance of the knowledge bases you can provide access to. If you need tweak the results further, the retrieval quality and curation of the task research report are additional performance levers. Finally, the end report generation must be customized for every use case. The required output format will differ vastly across knowledge tasks. The curation of the final report is crucial, especially when exposing the Deep Researcher to end users. You can generate the best and most relevant individual task research results, but if the final report does not fit your users need, the Deep Researcher will be useless as a whole.

Of course, we are living in the age of LLM agents. The implementation we discussed follows an LLM workflow instead of relying on an agent to make decisions. In most cases this this will be sufficient. We could consider implementing an agentic loop for the most complex tasks. The task research could, for example, iterate over every task research report until the research is thorough enough. However, agentic capabilities often overcomplicate architectures that perform just as good (or better) using a workflow. Thus, we leave the agent implementation as material for the next blog.

What is your experience with the Deep Research Pattern? Were you able to customize it? Which task in your daily work are you going to automate with it?

Artificial Intelligence | Generative Ai Tools | Typescript | Productivity | Workflow

Published in Level Up Coding

234K followers · Last published 11 hours ago

Follow

Coding tutorials and news. The developer homepage gitconnected.com && skilled.dev && levelup.dev

Written by Jakob Pörschmann

515 followers · 9 following

Follow

Building Gen AI stuff @ Google - https://hello-jp.net

Responses (6)

Griffin

What are your thoughts?

Souradip Pal he/him
2 days ago

Jakob, this Deep Research design pattern is a game-changer! Your step-by-step approach makes it so clear and implementable—awesome work!

11 1 reply Reply

Yaware Timetracker
15 hours ago

Jakob, this is a full blueprint, not just a writeup. One AI team I know connected their Deep Research workflow to Yaware.TimeTracker - it filtered noise from useful prompts by showing where engineers actually lost focus. The result - tighter task… more

3 1 reply Reply

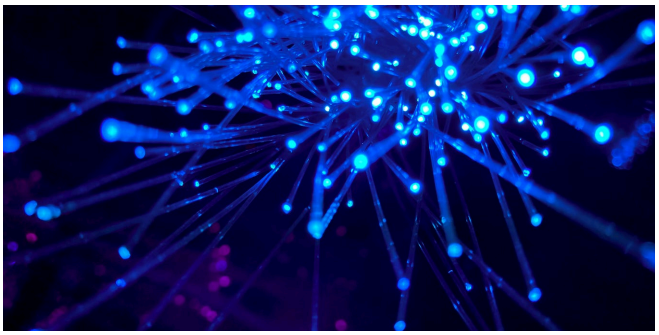
New Trend Computer Networks
1 day ago

This is next-level innovation! Automating tasks with a custom-built deep research system is a game-changer for efficiency and productivity.

4 1 reply Reply

See all responses

More from Jakob Pörschmann and Level Up Coding



In TDS Archive by Jakob Pörschmann

Graph RAG into Production—step-by-step

A GCP native, fully serverless implementation that you will replicate in minutes

Sep 23, 2024 888 7

In Level Up Coding by Anmol Baranwal

The guide to MCP I never had

AI agents are finally stepping beyond chat. They are solving multi-step problems,...

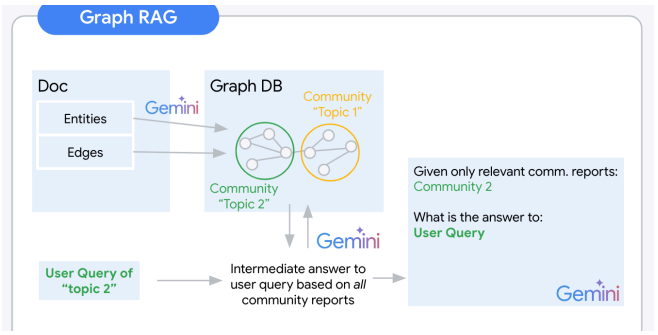
Apr 28 1.2K 16

In Level Up Coding by Varsha Das

How Senior Engineers Think Differently: 10 Mental Models You...

The Mindset Shift from Junior to Senior Engineer

Mar 31 1.7K 26



In TDS Archive by Jakob Pörschmann

Graph RAG—A conceptual introduction

Graph RAG answers the big questions where text embeddings won’t help you.

Aug 22, 2024 679 5

See all from Jakob Pörschmann

See all from Level Up Coding

Recommended from Medium

In Coding Nexus by Code Pulse

Why You Should Build an MCP Server This Weekend

I haven’t seen an opportunity this wide open since the early days of mobile apps.

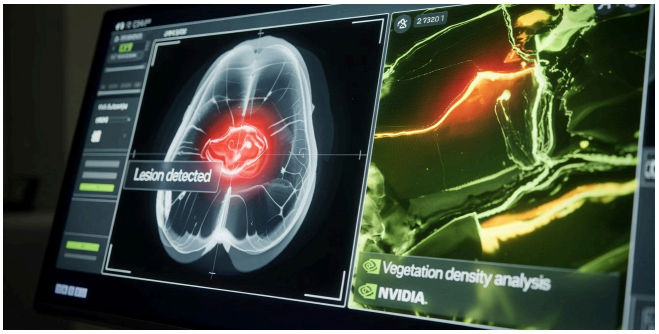
6d ago 598 10

Devlink Tips

Why I switched from obsidian: A real developer’s story and what I’...

Obsidian was great until it wasn’t. Here’s the real story, tools I replaced it with, and why yo...

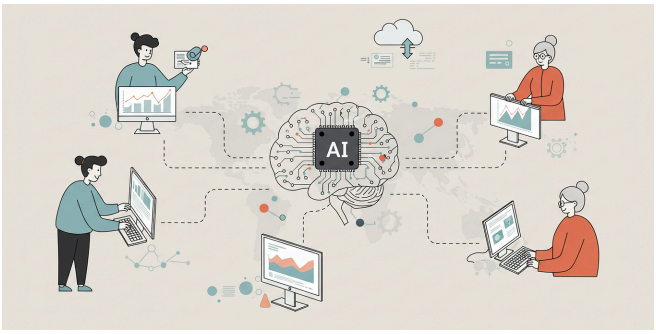
Apr 28 914 37



In Data Science Collective by Kajal Yadav

Nvidia Open-Sources ‘Describe Anything’—10 Real-World Uses...

From Diagnosing Diseases to Tracking Crops—How Nvidia’s Vision AI Is Changing the...



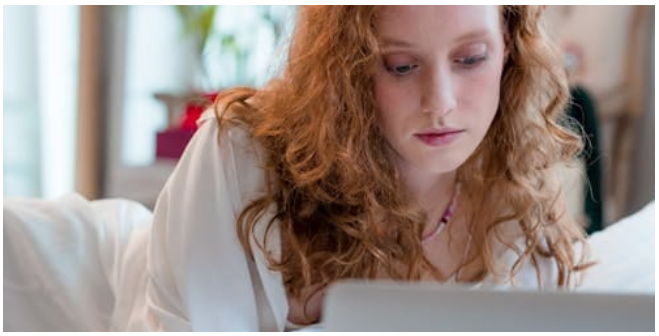
In AI Advances by Dr. Ashish Bamania

A Hands-On Guide To Federated Machine Learning With ‘Flower’

A deep dive into Federated Learning and training an ML model to detect eye diseases...

6d ago 349 6

4d ago 325 6



Krzyś

In The Startup by Divad Sanders

It Seems Obvious That AI Will Replace Zuckerberg Within 12 to 1…

In which we examine why those prophesying the end of coding jobs are unwittingly writin…

6d ago 749 23

4 Fast Startup Ideas You Can Steal and Launch Before Monday

Everyone’s playing with ChatGPT… you should build with it

Apr 29 792 27

See more recommendations