Internal RAG Platform Using Generative AI and Extension to Multi-Agent Orchestration
—Implementation Technologies, Operational Efficiency Outcomes, and Future Outlook

4.1 Introduction

AI technologies, especially large language models (LLMs), have made huge advances in recent years thanks to companies such as OpenAI, Google, and Anthropic, dramatically changing how enterprises use knowledge. At IIJ, we also began running an internal RAG platform called sbdGPT in the summer of 2023. This article discusses the background to its development, our objectives, and the system architecture, as well as its extension to multi-agent orchestration and a proposal document generation tool integrated with the internal RAG platform.

4.2 Background to and Objectives of Developing an Internal RAG

IIJ provides over 100 services for business customers and has amassed a vast and diverse information base in doing so, encompassing manuals, knowledge-sharing sites, news, inquiry emails, and more. Because these information assets are distributed and managed across multiple different platforms, accessing them across systems and quickly searching for and retrieving the information needed has long been a major challenge for sales staff and engineers. As such, the time spent repetitively querying internal sources and discovering information has hindered operational efficiency.

Against this backdrop, we began developing an internal RAG platform with the goal of collecting and processing document data scattered throughout the company, consolidating it into a central database, and using generative AI to provide specific answers in accord with user questions.

4.3 Internal RAG Architecture and Data Optimization

The internal RAG platform integrates multiple data sources, with the data being converted into vector embeddings for storage in a database. With the exception of the LLM and embedding models, we use IIJ services and open-source software to keep costs down while enabling independent, flexible development iterations and operations management.

Figure 1 provides an overview of the architecture.

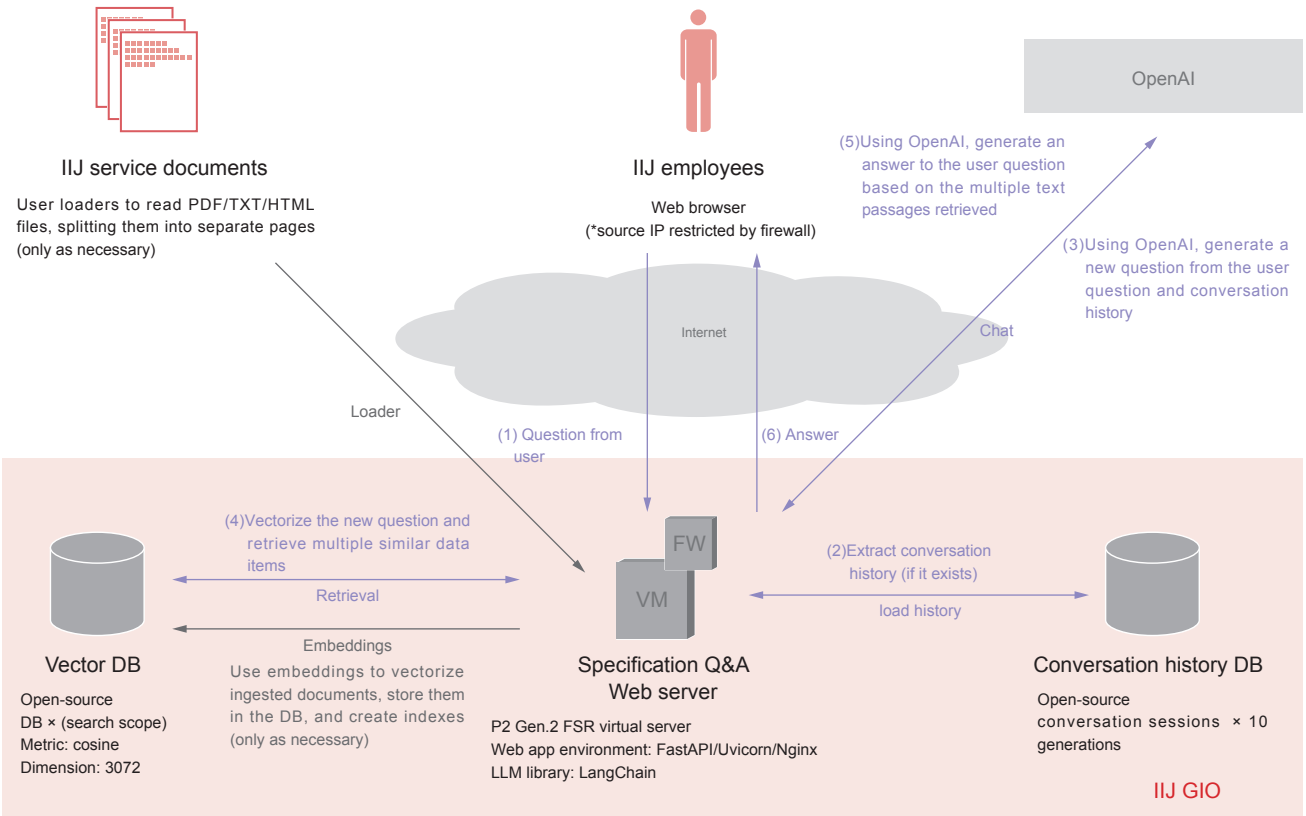


Figure 1: Overview of the Internal RAG Architecture

• Components

Knowledge base (IIJ service documents)
- Service detail materials (PDF files)
- Online manuals (HTML files)
- Internal technical QA emails (message files)
- Knowledge-sharing sites (HTML files)
- News articles (HTML files)

Web server platform
- IIJ GIO Infrastructure P2 Gen.2 Flexible Service VMs
- IIJ GIO Infrastructure P2 Gen.2 perimeter firewall
- FastAPI (OSS)
- Uvicorn/Gunicorn (OSS)
- Nginx (OSS)

Generative AI / Embedding models
- Models provided by OpenAI

LLM development framework
- LangChain (open-source version)

Vector DB
- ChromaDB (open-source version)

Conversation management DB
- PostgreSQL (open-source version)

*Except for the generative AI and embeddings, IIJ service infrastructure and open-source software are used.

Each knowledge dataset stored in the vector database targets files (PDFs, HTML, etc.) distributed across multiple platforms (Figure 2). We use the loaders LangChain provides for each file format to extract the content as text.

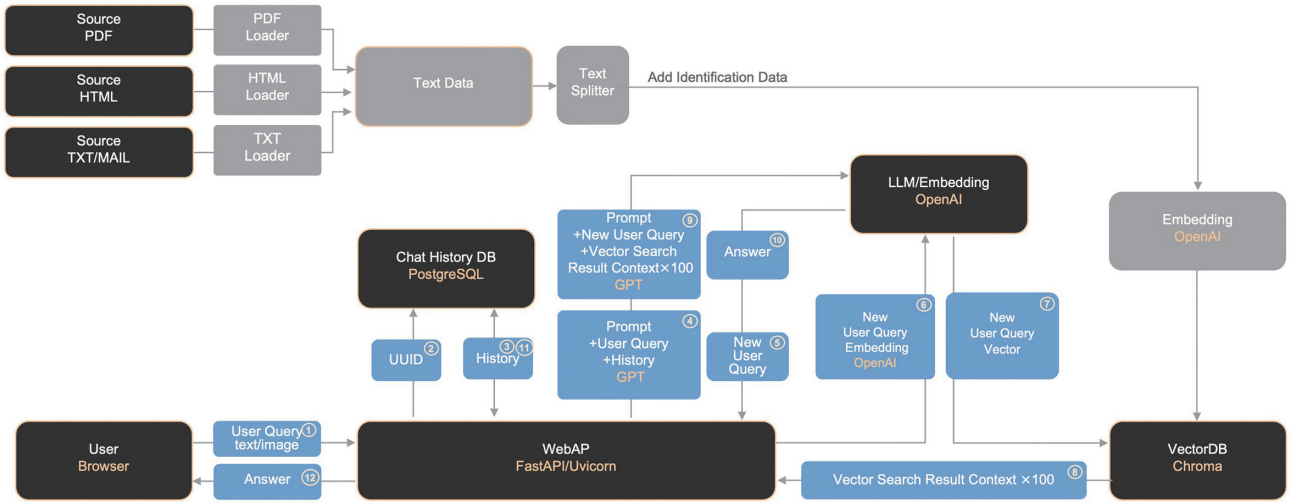


Figure 2: Operational Flow for the Internal RAG

Initially, we used the common approach of splitting the large volume of extracted text into appropriate lengths (chunking) using text splitters and stored it in the vector database in that form. In many cases, however, this approach failed to produce the expected answers to user queries.

Our investigation revealed that this was due to service documents being meaningfully (semantically) complete at the file level (PDF, HTML, etc.), such that short passages of text extracted in piecemeal fashion (e.g., chunks of data around 200 characters long) lacked sufficient contextual information about what was being said about which particular service. So semantic similarity was not being fully harnessed by the vector searches, and relevant information was not being properly retrieved as a result.

To address this, when chunking we now generate identification information from existing metadata contained in each file, such as the service name and title information, and attach it to each chunk. This has improved vector-search hit rates and answer accuracy.

4.4 Fact-Checking Measures

AI-generated answers may contain hallucinations (fabricated information). So fact-checking (verification of sources) is crucial when using AI-generated answers for business purposes. To ascertain what information the AI referenced when generating an answer, it is important to have the AI provide citations, and users need to check those citations to ensure the accuracy of the content.

Usability takes a big hit if the fact-checking process is complicated, however. So we built a mechanism into the tool to allow access to the data sources with minimal effort. Specifically, we implemented a feature that lets users bring up the source of the knowledge data (PDF, HTML, etc.) simply by clicking a citation link button displayed at the end of the generated answer.

Non-HTML data sources are stored on the Web server ahead of time and served statically via the browser, enabling immediate access to sources such as PDFs. This simplifies the user flow for determining the reliability of AI answers.

The identification information attached to each chunk also includes the data source file's FQDN (including page numbers in the case of PDFs). In the system prompt, we therefore specify a citation output format, as shown below, so that sources are presented immediately below the generated answer. This means users can simply click a button to display the citation details as text and instantly access the source file on the Web server.

```
# Example citation output format
<div hidden id="{{ id }}">
 <p class="cite_title">Cited Context 1</p>
<a href="{{ source file FQDN }}" target="_blank">{{  source file FQDN }}</a>
 <p>{{ context text }}</p>
 </div>
 <button class="open-button" onclick="refOpen('{{ id }}')">Main citations</button>
```

4.5 Business Efficiency Outcomes and Multi-Agent Extension of RAG Platform

Since we began using the internal RAG platform in the summer of 2023, usage over its roughly two years in operation has expanded to the point where around 30,000 queries are processed per month. This is helping achieve internal business efficiency gains on the order of 1,500 hours per month.

With the system in operation for some time, however, it also became clear that there are limitations to answers based on internal information using RAG alone. In particular, we found that in an increasing number of cases, it was difficult to provide appropriate answers without referring to information on the Internet such as the latest market trends, competitor comparisons, and differences in specifications vs. third-party products. The need to go beyond simple specification checks to provide higher-level decision-making support,

such as market research and trend analysis by sales staff and engineers, also started to emerge.

To address these issues, we extended the internal RAG platform and implemented a multi-agent architecture in which multiple specialized agents collaborate to generate answers. The conventional RAG mechanism vectorized the internal data (IIJ service manuals, knowledge-sharing sites, inquiry emails, etc.) and used generative AI to produce answers, but handling queries requiring external information, such as comparisons of IIJ's service specifications with those of other companies or those dealing with proposals based on the latest technology trends, was problematic.

To resolve this constraint, we refactored the internal RAG platform and built a multi-agent platform. The main agent automatically calls other agents as needed based on routing logic, such as the IIJ Service Agent, an agentized version of the internal RAG platform, and the WebSearch Agent, which retrieves the latest external information.

In terms of the actual control structure, we use a state-transition graph managed by LangGraph to orchestrate inter-agent collaboration. The main agent analyzes the question text, determines whether it can be solved with internal data or whether external information is required, and then calls the relevant subagent(s). The subagent results are returned to the main agent and ultimately presented to the user as an integrated response. This makes it possible to generate comprehensive answers combining internal information with up-to-date external information.

The subagents also support parallel execution, enabling simultaneous processing across multiple information domains, thereby producing results at speeds impossible to achieve manually. In addition, we provide enhanced transparency via visualizations of the LLM's inference process. The UI successively streams information on which agents were invoked and which information sources were used to generate the answer, allowing users to track the LLM's reasoning process in real time.

As a result, we were able to extend the answer coverage to areas over which the existing internal RAG platform could not provide full coverage, helping to greatly improve the user experience. The internal agent platform built on multi-agent principles overcame the information shortage issues that were difficult to address with the single-agent approach and represents an important step in significantly expanding the scope of AI utilization.

4.6 Combining Deep Research and RAG

Google's Deep Research, announced at the end of 2024, drew attention for its ability to autonomously search vast online information sources and analyze and integrate what it finds, enabling it to perform multi-step investigations much like a researcher or analyst would. In February 2025, OpenAI also released its own Deep Research offering, and in June of the same year, it made this available as the Responses API, providing practical components that can be incorporated into existing agent environments.

The defining feature of Deep Research is its ability to autonomously iterate through a multi-step cycle of planning, searching, scrutinizing, integrating, and generating. It executes multiple web searches, recalibrates its plans based on verified data, and ultimately delivers a highly reliable summary report. While some tasks can therefore take around 30 minutes to complete, it is a powerful tool for research in domains requiring a high level of comprehensiveness and evidence, such as technical literature reviews, competitor and market research, and regulatory comparisons.

Because our internal agent platform already used a multi-agent architecture based on LangChain and LangGraph, adding Deep Research as a subagent was relatively straightforward.

Specifically, we defined a Deep Research agent with @tool and implemented a function that calls the Responses API inside it. When a user query is received, Deep Research iteratively searches the web and, as needed, goes through several stages of reasoning and integration to return a structured response that includes a report, data tables, and a list of sources.

In addition, we introduced a mechanism that puts a ClarifyQuery (query clarification) agent at the front end of Deep Research. This prevents ambiguous queries from being passed directly to Deep Research, which would increase search retries and result in ballooning costs and latency. This agent reviews the user's input and, if necessary, poses follow-up questions to clarify the query before executing Deep Research. This facilitates comprehensive and efficient research that is aligned with user intent.

We also made the inference parameters provided by the Deep Research API (summary, effort, verbosity) adjustable by users, allowing them to select the depth of investigation and output volume according to their use case. For example, for a concise executive summary, users can select effort = low and verbosity = low, while for careful investigations such as specification comparisons, they might specify effort = high and verbosity = high. This allows for flexible control over output granularity for the same query with just a few clicks.

The UI also provides visualizations of processing progress and cost. In addition to streaming the AI's inference process, we also display the total number of tokens used and model consumption costs so that users can see the cost incurred in generating an answer. As developers, meanwhile, we can monitor token consumption and cached token usage, so we are able to use this information to optimize the tool.

A new consideration that emerged in the process of introducing Deep Research was the need for autonomous research capabilities that also include internal data. In its standard form, Deep Research is designed primarily for publicly available information on the Internet. It did not provide a mechanism for directly integrating with all internal knowledge bases. While we thought it would be technically feasible to connect internal knowledge sites to Deep Research via the APIs provided, this turned out to be unrealistic because of the high non-technical hurdles, namely the need for internal coordination, agreement, and approvals.

So our initial approach was to combine the IIJ Service Agent (internal RAG) and the WebSearch Agent, with the processing steps finely controlled via prompts. With the WebSearch Agent exploring the latest information online while the IIJ Service Agent (internal RAG) searches through internal information, this made it possible to ultimately return integrated results to the user. But with earlier LLMs (GPT-4.1 and earlier), ensuring sufficient comprehensiveness and consistency was difficult due to limits on reasoning steps and weak tool-invocation control.

Accordingly, we adopted as the main agent GPT-5, which was released during our development verification process and claimed to be optimized for agentic tasks. GPT-5 offers enhanced capabilities in terms of tool invocation, instruction following, long-context understanding, and reasoning retention across tool calls, greatly improving interoperability with tools in agentic workflows. As a result, by invoking each subagent multiple times to collect comprehensive

information, we were able to generate optimal answers based on roughly 30 to 80 citations, including internal data.

This represented a significant step forward as we were now able to integrate both internal data and external information sources to provide highly comprehensive, reliable answers even to queries for which we would previously have determined information to be insufficient. And as developers, the ability to resolve the issue simply by dropping in the latest LLM without any major changes to existing architecture was key and really brought home the extensibility of the agentic platform and the potential for its ongoing evolution.

4.7 Developing a Proposal Document Generation Tool

Once operation of the internal RAG platform had stabilized, we started developing a new web application called Panorama, a proposal document generation tool. When creating PowerPoint-format proposals in accord with given requirements, sales staff and engineers were spending a lot of time on the task of selecting and editing slides from internal templates. Panorama was developed to solve that problem. It provides a simple interface for doing everything from requirements input through to automatic generation of a proposal draft (Figure 3).

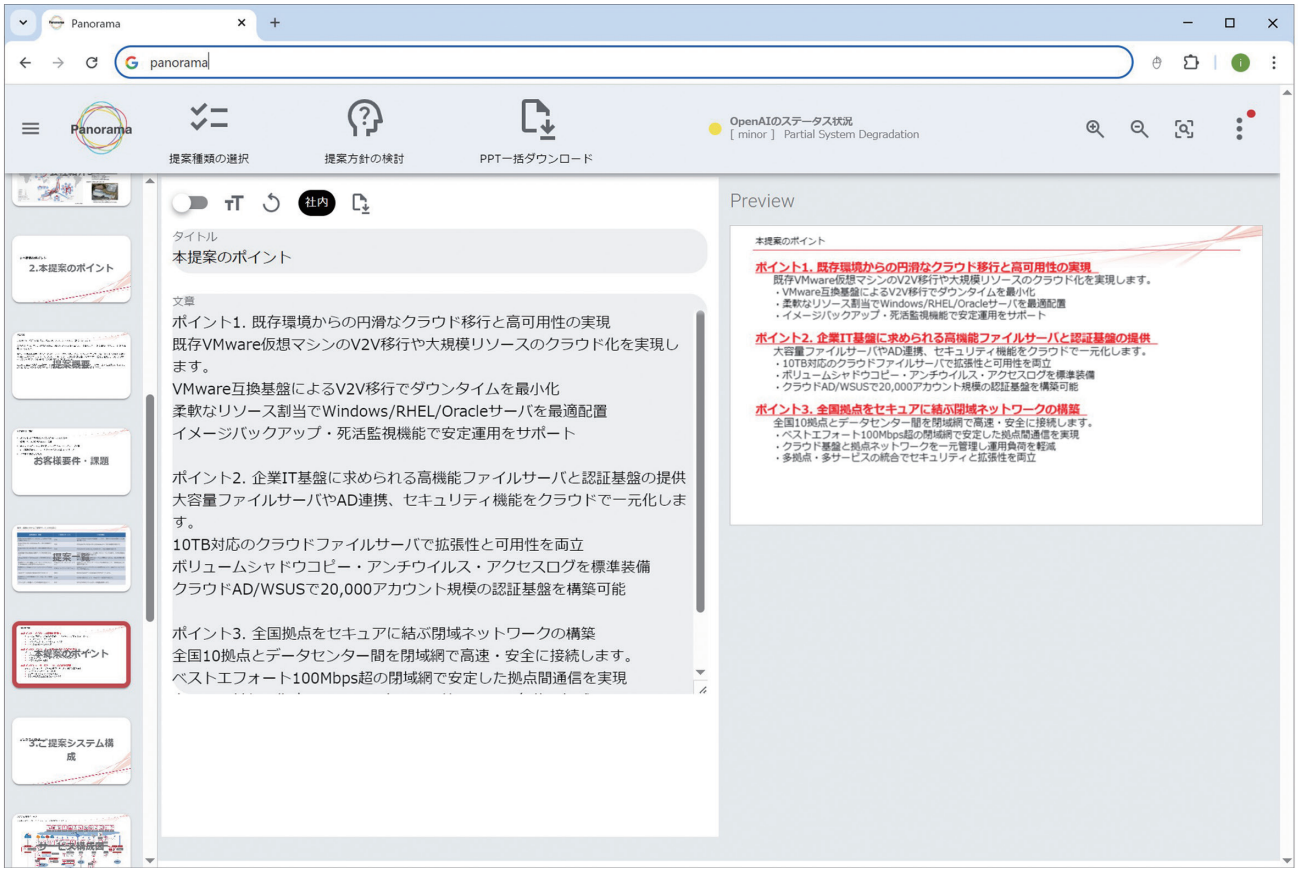
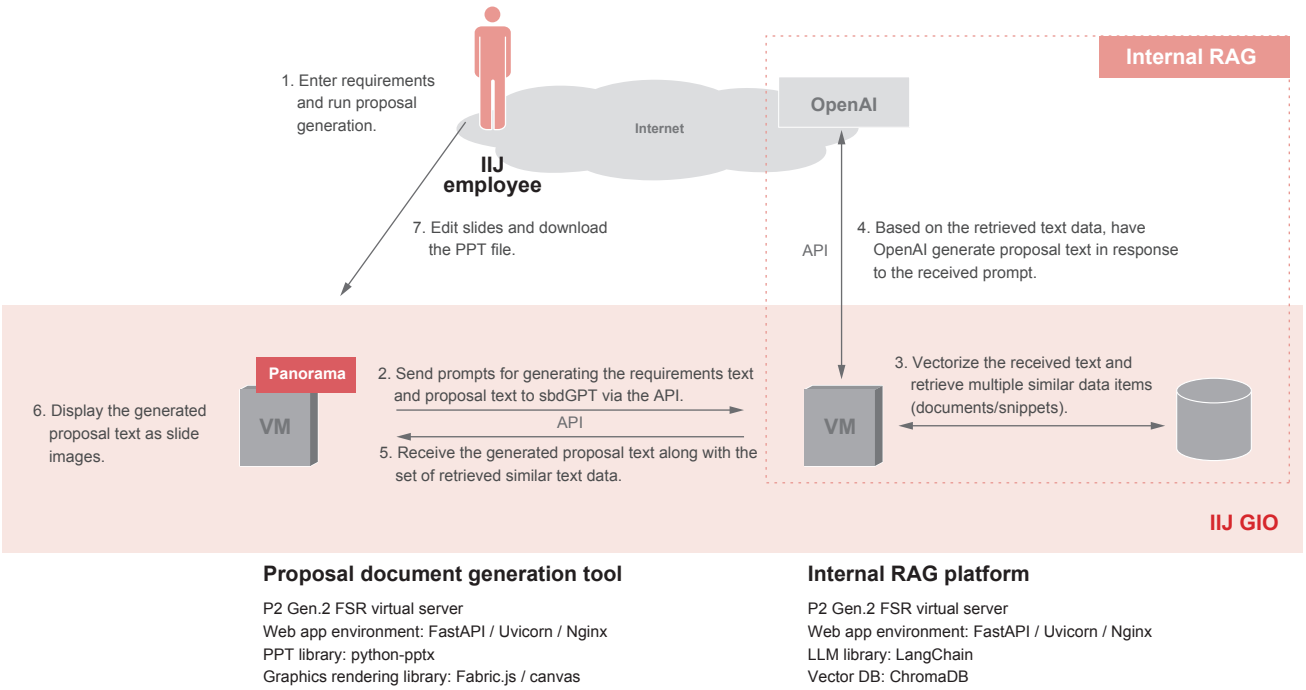


Figure 3: Proposal Document Tool's User Interface

The proposal document generation tool works as follows. After the user enters requirements and desired services via the browser and submits the request, the Web server then calls the internal RAG platform API. Using the input data and the system prompt, the tool automatically generates the most appropriate service proposal text (Figure 4). The generated output is converted into JSON describing each slide's structure and layout, and this is reflected in the browser-based editor and rendered in real time on an HTML5 Canvas for preview purposes. Fabric.js is used for rendering and editing, and users are able to review and revise the architecture diagram layout and proposal text in the browser before downloading the final output as a PowerPoint file.

In technical terms, the system consists of a frontend built with Vue.js + fabric.js and a backend built with FastAPI + python-pptx. On the frontend, edits to slides and architecture diagrams are reflected in real time, and all edit data is centrally managed in JSON format. The data are sent via FastAPI and used to generate PPTX files on the backend. On the backend, Python is used to manipulate and update XML configuration files corresponding to template slides based on the JSON data, and based on the resulting modifications, the final proposal document file is generated using the python-pptx library.

Through these mechanisms, the AI-generated proposal text and automatically plotted architecture layout diagrams are reflected in the slides, enabling users to create consistent-quality proposals in a short amount of time with minimal editing work. Integration with the internal RAG platform also means that proposal text automatically reflects internal



Proposal document generation tool

P2 Gen.2 FSR virtual server
Web app environment: FastAPI / Uvicorn / Nginx
PPT library: python-pptx
Graphics rendering library: Fabric.js / canvas

Internal RAG platform

P2 Gen.2 FSR virtual server
Web app environment: FastAPI / Uvicorn / Nginx
LLM library: LangChain
Vector DB: ChromaDB

Figure 4: Overview of Proposal Document Tool System Architecture

knowledge such as IIJ service specifications, the rationale for recommendations, and key proposal points. Source information is displayed clearly on the editing screen, and users can instantly confirm data sources at the click of a mouse, making it easy to fact check proposal content.

Going forward, we plan to incrementally add template slides and add more features such as PDF-format exports and enhanced diagram editing features.

4.8 Looking Ahead

The evolution of the latest language models and agent technologies is making the autonomous execution of complex tasks a reality, and agentic systems designed to support business automation and efficiency improvements are growing rapidly in importance.

For now, we plan to continue expanding the capabilities of the AI agent platform, progressively developing and adding tools designed for specific tasks. We will also examine mechanisms for creating a mutually reinforcing cycle between incentives for the creators of referenced data sources and increases in citation frequency.

Over the medium to long term, we aim to transition to an architecture that uses local LLMs and constitutes a complete system within IIJ's own infrastructure, with our goal being to develop an environment that we can offer to customers as a package integrated with IIJ's wide range of services.

Kazunori Ebine

AI Platform Promotion Office, System Development Division, Network Services Business Unit, IIJ
Mr. Ebine joined IIJ in 2005. He worked on building systems for the public and private sectors, handling a wide range of responsibilities from requirements definition through design, implementation, and operation. He has contributed to revenue growth by proposing and rolling out cloud services and building technology delivery structures. Having previously led AI-driven efficiency improvements and cross-departmental initiatives as a Deputy General Manager and a Manager, he is currently spearheading the development and planning of AI platforms.