



i2ai / i2store Walkthrough

Dr.-Ing. Boris Horner

What i2ai / i2store does

- Extensible middleware for AI models (LLM, speech-to-text, image processing, ...).
- Extensible middleware for arbitrary transforms and I/O with other systems (RDBMS, CMS, CRM, ERP, ...).
- Sophisticated API with login, exposing **all** functions of the server.
- Integrate and expose the AI-optimized storage engine **i2store** combining metadata, graph and full-text search with vector ranking.
- Provide a web GUI for administration and custom plugins.
- User / group / permission management.
- Generic prompt editor and process editor and runtime, enabling power users to build new use cases from reusable, elementary functions.
- Using the metadata, relation, text and vector management, powerful Content Delivery Portals can be easily built by adding a GUI.

What i2ai / i2store does not

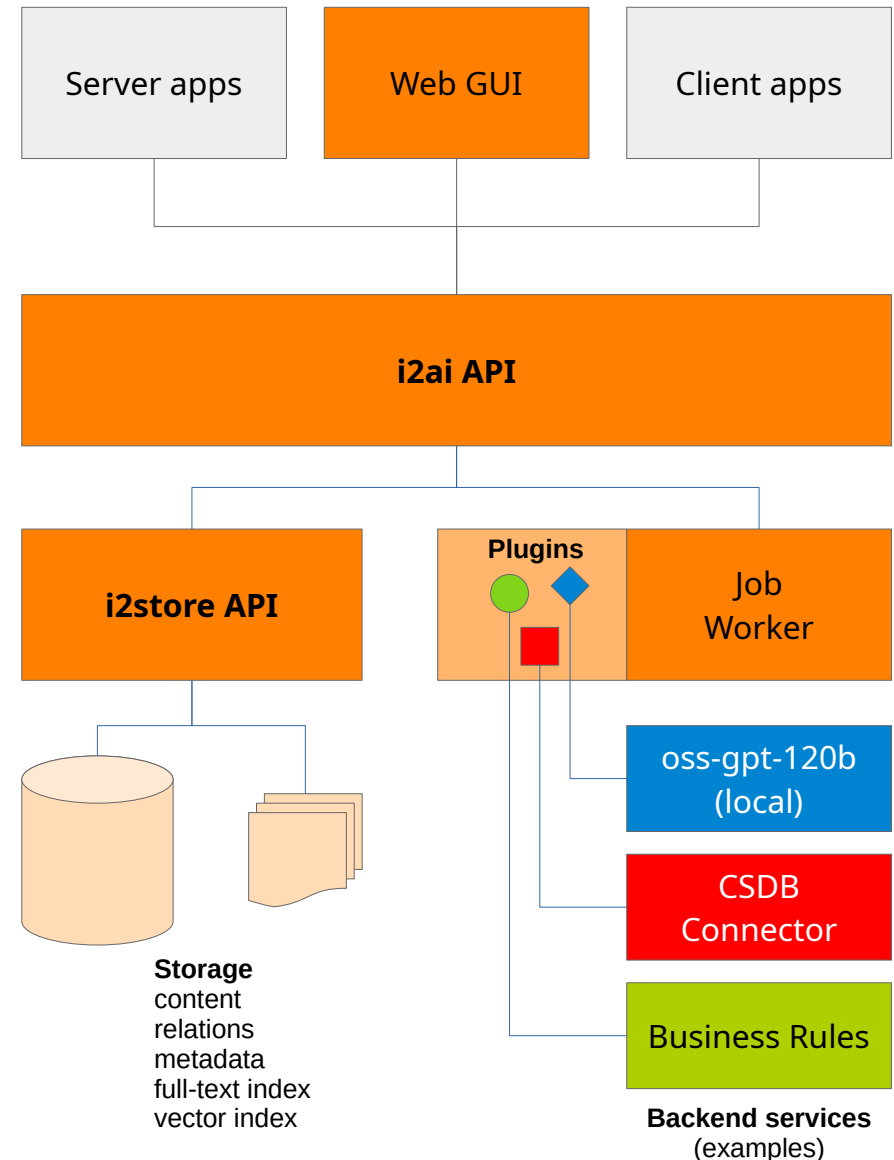
- **i2ai** neither provides its own AI models nor does it offer training or fine-tuning infrastructure; local models can be hosted in the network and used by **i2ai**.
- **i2store** is not meant as a replacement for authoring environments, even though it could be used. It is perfect for managing AI knowledge, CDP data etc. with deep semantics.

i2ai API

- The core part of the system is the API that can be addressed by standard HTTP.
- The API exposes commands as needed, to enqueue requests and fetch results etc.
- Storage is used by the API and exclusively accessible through the API.
- API use is restricted by a token; optionally user management and permissions.

Job Worker

- The Job Worker executes enqueued tasks and sets the status to ready (or failed) when done.
- The various atomic functions are provided as plugins; new plugins can be added.
- Plugins can have dependencies like local or remote AI models or server apps.
- Concurrency behaviour and capabilities can be configured per resource.





i2ai / i2store expose their entire functionality under a complete API. There is a standard web application implementing GUI for all administrative tasks and a plugin interface for custom GUI, but the frontend app only takes input and displays results.

Thus, all functions are available in the standard GUI, but generally will be integrated elsewhere, in addition to the web GUI or exclusively:

- New, custom web interface (with or without login)
- Extension of an existing web application
- Client tools like XML editors, review frontends, ERP clients, ...
- Server processes

What the user sees in detail is determined by the integration.

In our example, we'll show a simple RAG application based on documentation for the medical imaging standard DICOM (Digital Imaging and Communications in Medicine). We'll use the standard web application with a specific RAG viewer plugin.

Other applications (like business rules validation in defense, technical authoring support, data analysis and anything else) can be built using the same techniques.

RAG stands for „Retrieval Augmented Generation“.

That means, the Large Language Model (LLM) does not respond (**generate**) directly to the user's question. Before it answers, the process **retrieves** matching content and uses it for a fact-based, improved (**augmented**) response.

Typical RAG process (AI-based steps: ■)

- Translate user input to search terms for matching data ■
- Execute the search and retrieve matches with URLs and summaries
- Rank search results according to summaries / remove irrelevant ones ■
- For all remaining results:
 - ♦ Fetch content
 - Filter content to leave only the sections relevant to the user input in place ■
 - ♦ Join all results in a JSON structure
- Answer the user input based on the retrieved content ■

Later, we'll see how such a process is modeled.

RAG is only one example – any kind of functionality can be built.

Most people use AI in their daily life and know how helpful it can be, and wish to use it for their professional tasks, too.

So decision-makers basically have three options with respect to AI use:

1. Tolerate employees using AI services they are familiar with.
2. Restrict 1.
3. Analyze the internal processes and provide controlled AI infrastructure.

Option 1 is undesirable, because every work day, confidential information and intellectual property go where they shouldn't go („Shadow AI“), and the models and prompts used are totally uncontrollable.

Option 2 (if it can be 100% implemented) solves the Shadow AI issue at the cost of efficiency.

Option 3 fulfills the need for AI based automation, but data protection and standardization of prompts and other process steps are **by design**.

Again, all this is not only true for RAG, but for any AI process.

Question
How do I troubleshoot a communication problem between host and application?

Result
Based on the provided background information, troubleshooting a communication problem between a host and a hosted application involves checking the state management and interface methods. The document describes the Application Interface with methods that can help diagnose issues:

Key Troubleshooting Steps

- 1. Check Application State with `getState()`**
 - Call `getState()` to retrieve the current state of the hosted application. This can be used to verify if the application is still running and in a valid state (e.g., IDLE, INPROGRESS, EXIT).
 - If the application has exited without calling `notifyStateChanged()`, this method can detect that the application is no longer operational.
- 2. Verify State Transitions with `setState()`**
 - Use `setState(newState)` to request a state change. The method returns `TRUE` if the request is valid and accepted, but note that the actual state change may be asynchronous.
 - If `setState()` returns `FALSE`, it indicates the requested state transition is not allowed from the current state, which may point to a communication or state mismatch.
 - Monitor for `notifyStateChanged()` callbacks to confirm the state change occurred.
- 3. Test GUI Visibility with `bringToFront()`**
 - Call `bringToFront(requestedScreenArea)` to check if the application can respond to requests for visibility and focus. A `TRUE` return indicates the application received the request.
 - If the application has no GUI (headless), it should always return `TRUE`. A `FALSE` return may indicate the application is not in a responsive state (e.g., not IDLE or INPROGRESS).

Additional Diagnostic Notes

- The `getState()` method can be called at any time and serves as a "heartbeat" to confirm the application is alive.
- If `setState()` is called multiple times, the application will abort the previous request and process the latest one—this can help resolve stuck states.
- Communication problems may manifest as unexpected `FALSE` returns from these methods or missing `notifyStateChanged()` callbacks.

Sources [2]

- [8 Interfaces](#)
- [PS3.17](#)

- **Question:** What the user has typed in.
- **Result:** User input answered by the LLM.
- **Sources:** Clickable links to the data modules used.



i2ai Administrator

RAG

- Wikipedia
- DICOM
- Content**
- Nodes
- Administration**
- Attribute fields
- Taxonomies
- Contexts
- Domains
- Formats
- Data types
- Node types
- Relation types
- Resource groups
- AI models
- AI prompts**
- Transform plugins
- Transforms
- Processes
- Users
- Groups
- Permissions
- Jobs
- Tokens

Id	Name	Label	Model
4	answer_rag_input_deepseek	Answer RAG input (DeepSeek)	deepseek-chat
3	condense_html_deepseek	Condense HTML (DeepSeek)	deepseek-chat
1	get_google_query_deepseek	Get Google search strings (DeepSeek)	deepseek-chat
6	get_google_query_gpt_oss	Get Google search strings (gpt-oss)	gpt-oss-120b
2	rank_google_results_deepseek	Rank the results from Google by relevance (DeepSeek)	deepseek-chat
5	test	Test prompt	gpt

Id 6

Name get_google_query_gpt_oss

Label Get Google search strings (gpt-oss)

Model gpt-oss-120b

Description
Based on a user's question in a RAG system, the prompt returns google search strings

Parameters

max_tokens 2000

temperature 0.5

top_p 1.0

Prompt Definition

system / static
Your task is one step in a RAG system. Based on the user input, create a google search string, typically consisting of multiple terms. The google search is already restricted to the URL containing relevant content and should retrieve results relevant to answering the user input. This is the user input.

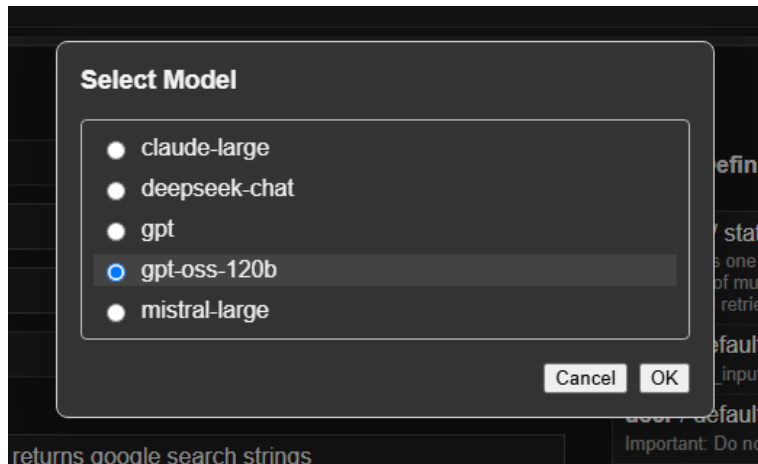
user / default / field
Field: user_input | Formats: text/plain, text/markdown

user / default / static
Important: Do not add markup, do not explain the result. Just return the search string as plain text.

Outbound Interface

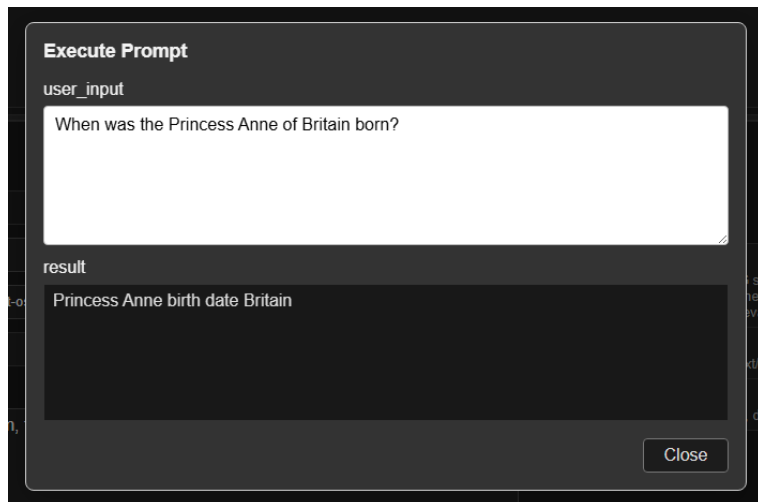
result
text

Start with building prompts. A prompt consists of static and field messages that can be added and edited as needed. Parameters can be changed with sliders.



Model support

- Models are configurable and extensible.
- Custom models can be integrated with custom model adaptors.
- Choosing or changing the model a prompt uses is just one mouse click.



Immediate testing

- Directly from the prompt editor, prompts can be tested.
- Prompts with multiple field messages can also have multiple inputs.
- This prompt generates a Google search string from a user input.

i2ai Administrator

RAG

- Wikipedia
- DICOM
- Content**
- Nodes
- Administration**
- Attribute fields
- Taxonomies
- Contexts
- Domains
- Formats
- Data types
- Node types
- Relation types
- Resource groups
- AI models
- AI prompts
- Transform plugins
- Transforms**
- Processes
- Users
- Groups
- Permissions
- Jobs
- Tokens

Id	Name	Plugin	Resource Group
4	dicom_web_html_fetch	web_html_fetch	puppeteer
3	google_dicom_search	google_search	google
1	google_wikipedia_search	google_search	google
2	wikipedia_web_html_fetch	web_html_fetch	puppeteer

Id: 3

Name: google_dicom_search

Transform Plugin: Google search

Resource Group: google

Parameters:

```
{
  "parameters": {
    "google_search_id": [REDACTED]
    "google_search_key": [REDACTED]
  }
}
```

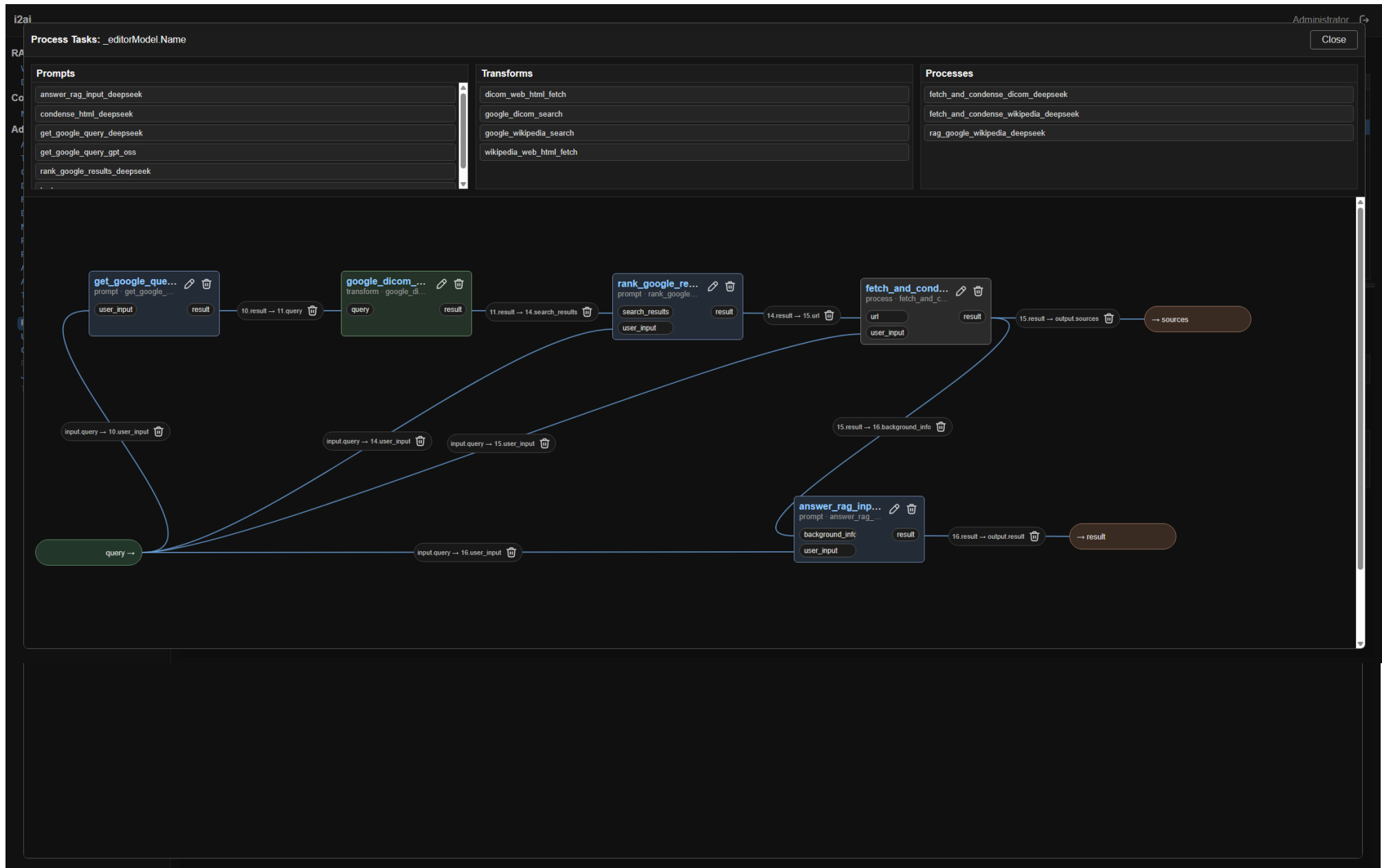
Inbound Interface

query
text

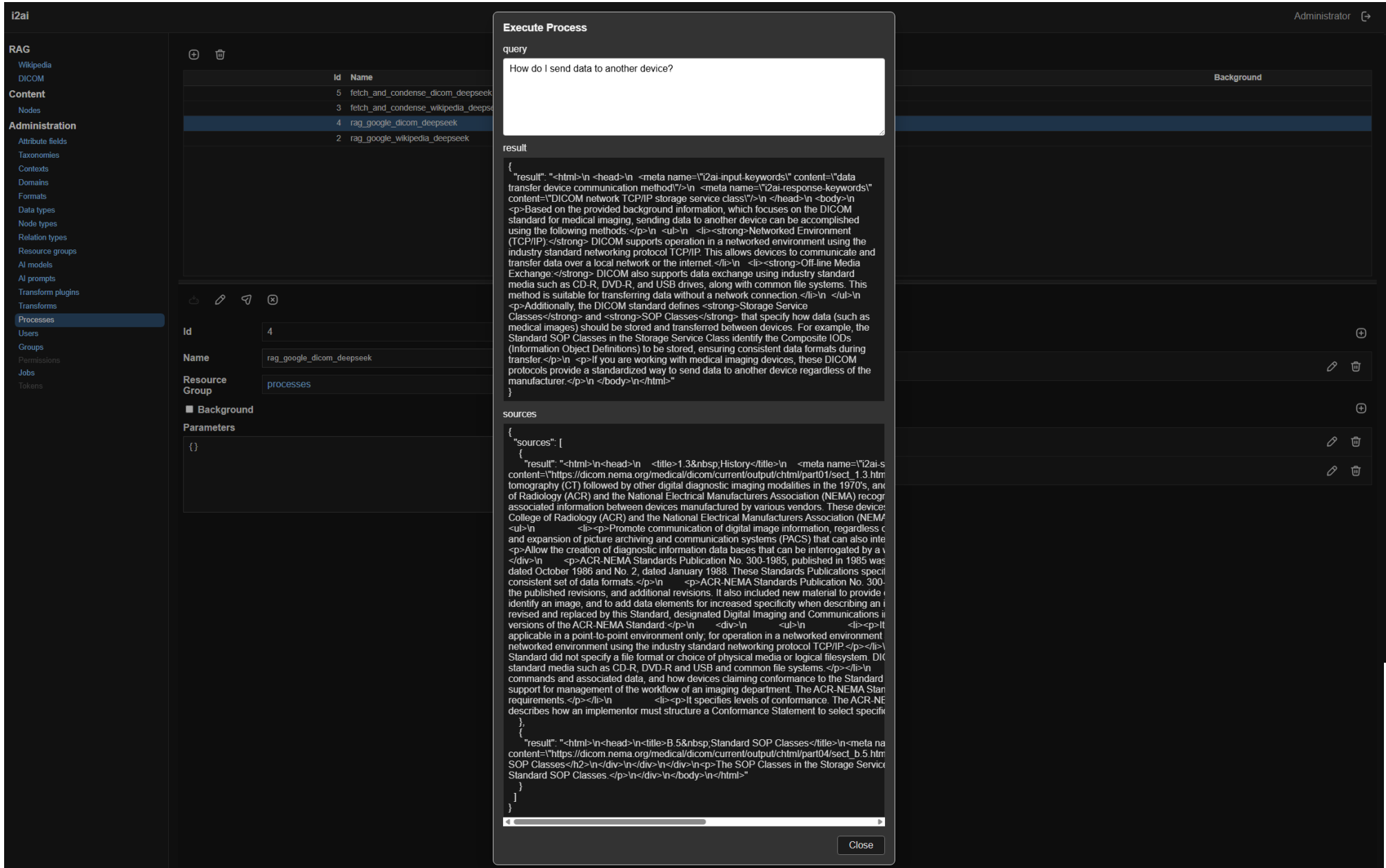
Outbound Interface

result
json

Transforms work much like prompts, have inputs and outputs, but do not employ an AI model. A transform can, e. g., read or write content, convert formats or access an API. This transform performs a search with Google Search API.



Prompts, transforms and subprocesses can be plugged together to processes.



The screenshot displays the i2ai application interface. On the left is a sidebar menu with categories like RAG, Content, Administration, and Transformations. The main area shows a table of processes. A modal window titled 'Execute Process' is open, showing a query, its result (HTML), and the sources used for the result.

Execute Process

query

How do I send data to another device?

result

```
{
  "result": "<html>\n<head>\n  <meta name='i2ai-input-keywords' content='data transfer device communication method'\n  <meta name='i2ai-response-keywords' content='DICOM network TCP/IP storage service class'\n</head>\n<body>\n<p>Based on the provided background information, which focuses on the DICOM standard for medical imaging, sending data to another device can be accomplished using the following methods:</p>\n  <ul>\n    <li><strong>Networked Environment (TCP/IP):</strong> DICOM supports operation in a networked environment using the industry standard networking protocol TCP/IP. This allows devices to communicate and transfer data over a local network or the internet.</li>\n    <li><strong>Off-line Media Exchange:</strong> DICOM also supports data exchange using industry standard media such as CD-R, DVD-R, and USB drives, along with common file systems. This method is suitable for transferring data without a network connection.</li>\n  </ul>\n<p>Additionally, the DICOM standard defines <strong>Storage Service Classes</strong> and <strong>SOP Classes</strong> that specify how data (such as medical images) should be stored and transferred between devices. For example, the Standard SOP Classes in the Storage Service Class identify the Composite IODs (Information Object Definitions) to be stored, ensuring consistent data formats during transfer.</p>\n<p>If you are working with medical imaging devices, these DICOM protocols provide a standardized way to send data to another device regardless of the manufacturer.</p>\n</body>\n</html>"
}
```

sources

```
{
  "sources": [
    {
      "result": "<html>\n<head>\n  <title>1.3&nbsp;History</title>\n  <meta name='i2ai-s content='https://dicom.nema.org/medical/dicom/current/output/html/part01/sect_1.3.htm tomography (CT) followed by other digital diagnostic imaging modalities in the 1970's, and of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA) recogr associated information between devices manufactured by various vendors. These device: College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA/ <ul>\n    <li><p>Promote communication of digital image information, regardless c and expansion of picture archiving and communication systems (PACS) that can also inte <p>Allow the creation of diagnostic information data bases that can be interrogated by a v </div>\n    <p>ACR-NEMA Standards Publication No. 300-1985, published in 1985 was dated October 1986 and No. 2, dated January 1988. These Standards Publications specif consistent set of data formats.</p>\n    <p>ACR-NEMA Standards Publication No. 300- the published revisions, and additional revisions. It also included new material to provide i identify an image, and to add data elements for increased specificity when describing an i revised and replaced by this Standard, designated Digital Imaging and Communications ii versions of the ACR-NEMA Standard:</p>\n    <div>\n      <ul>\n        <li><p>It applicable in a point-to-point environment only, for operation in a networked environment networked environment using the industry standard networking protocol TCP/IP.</p></li>\n      </ul>\n      Standard did not specify a file format or choice of physical media or logical filesystem. DIC standard media such as CD-R, DVD-R and USB and common file systems.</p></li>\n      commands and associated data, and how devices claiming conformance to the Standard support for management of the workflow of an imaging department. The ACR-NEMA Stan requirements.</p></li>\n      <li><p>It specifies levels of conformance. The ACR-NE describes how an implementor must structure a Conformance Statement to select specif


Close


```

Processes can be tested from the editor, too. That's all, just hook it under the GUI.



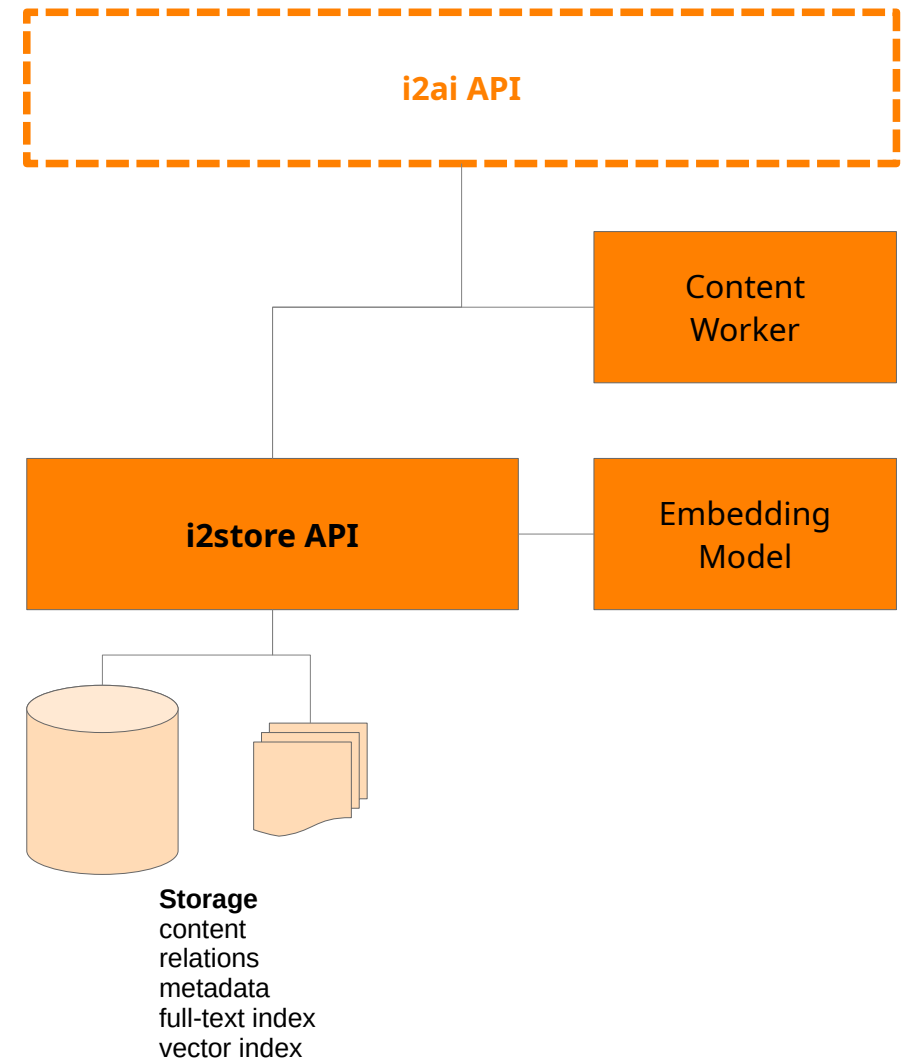
The RAG example we've shown does not use local storage at all, rather uses Google Search API and retrieves content from public web.

i2store API

- i2store API provides access to all content operations.
- It is wrapped by i2ai API and not directly accessed by clients.
- For vector indexing and search, an embedding AI model is running as an internal service.
- Content worker extracts text for indexing from PDF, XML, JSON and other formats (plugins).

Search

- Attributes have types and can use taxonomy.
- Fulltext search on all text content.
- Semantic relations can be followed.
- Vector ranking.
- All can be combined in one query.



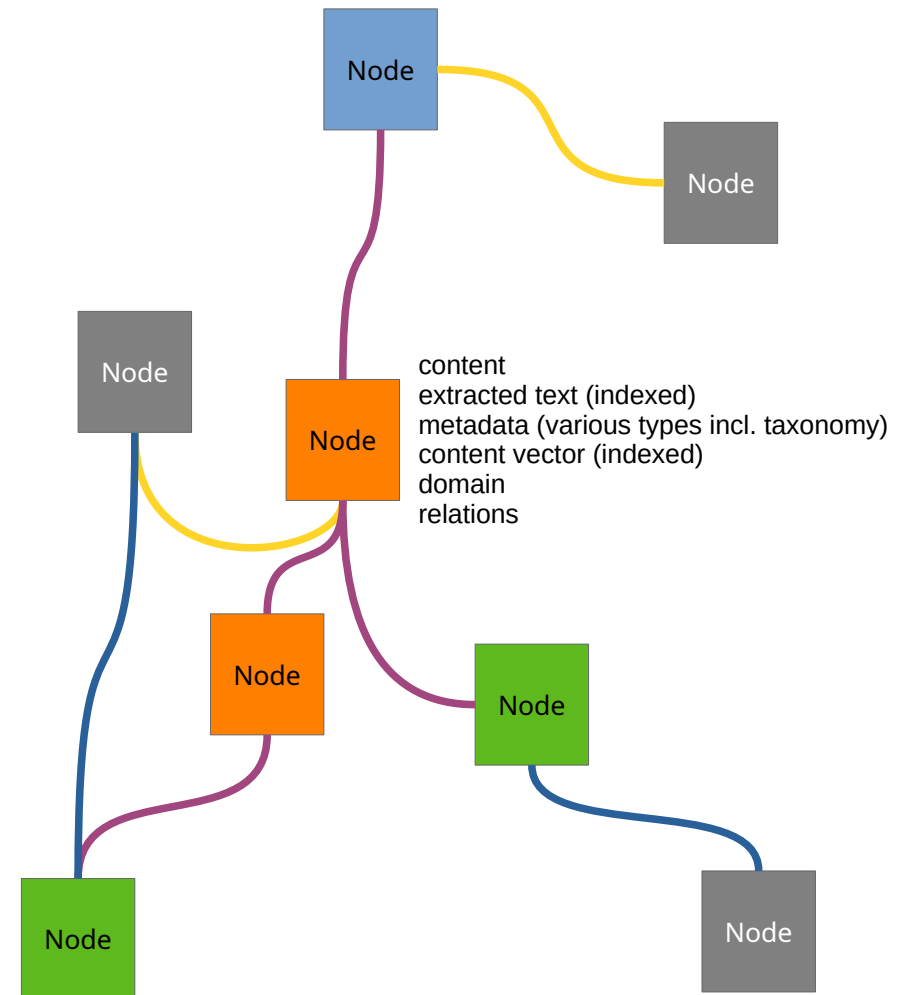
Each data object is represented as a **node**:

- Searchable **metadata**
- **Full-text** and **vector indexed** content
- Typed **relations** connecting nodes
- Relations can also have searchable metadata
- Every node belongs to exactly one **domain** for grouping / packaging
- All nodes in a domain have the same permissions (but read-only flag available)

Thus, the data model combines these axes:

- Metadata search in RDBMS
- Full-text index
- Vector index based on embedding model
- Relation tracking filtered by relation type and metadata

One search API call can query all four axes simultaneously.



Domains can serve as all types of containers on very different granularity, as needed. A domain could be used for:

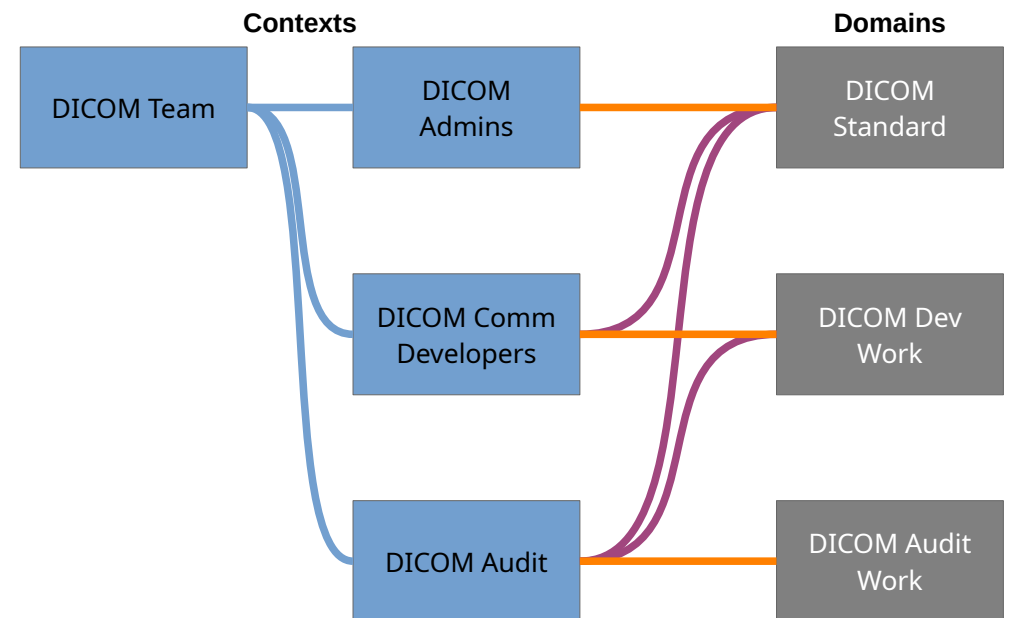
- Sets of LLM background data
- Upload packages for one product in a CDP
- Resources that have common access rights in general

Domains have an owner with full access. Otherwise, domains have no permission sets.

Contexts are the layer that (implicitly) assign permissions to domains.

Each context has a **root domain** with r/w (restricted by permissions) and optionally one or more read-only **secondary domains**.

Contexts can be **nested**.



Nodes have more properties helping organize the data:

- Read-only flag, so that certain nodes can be protected from overwriting in r/w domains.
- Format from an extensible list with MIME types and extensions.
- Node type as classification of the node purpose (image, topic, ...).
- Data type for mapping generic data properties that help modules decide whether they can work with the data or not.
- Several content fields:
 - Core content is stored in the DB (JSON, XML ...) or as a file (PDF, images, ...).
 - On write operations, full text is extracted for supported formats. More formats can be added as plugins. The full text gets indexed for search.
 - Once full text is available, a vector is generated by an embedding model and stored in the DB. The vectors are used for similarity ranking.
 - Preview field for optional HTML representation.
 - Summary field for optional, AI-generated abstract (can be used for ranking etc.).
 - Nodes have a grouping mechanism within the domain, so versions and variants (for example, languages) can be systematically managed.



We've discussed a RAG application in this presentation. In this basic form, RAG is just a one question – one answer pattern. This is particularly useful in no-login setups, since no data is stored and no user context needs to be known.

An obvious extension is a chat application that enables a dialogue between a user (or multiple users that can be invited) and the system (or multiple agents). Chats are persistent and can be picked up later – thus, the message history is saved.

Apart from that, uploaded files are saved in **i2store** as well as topics with gained knowledge. They can also be categorized with metadata, so they can be found and reused by the system more easily.

In a knowledge based chat, users can discuss technical or other questions with the system and reuse the knowledge gained from the dialogue.

Moving data like technical documentation from one infrastructure to another can often be formally automatized to some level, but many content-related tasks are usually manual.

Here are some examples for steps that can be partially or fully robotized:

- Taxonomy extraction
- Semantic specialization (e.g., topic → concept, reference, task)
- Metadata assignment
- Summary creation
- Re-modularization according to rules
- Authoring rules (how to write instructions, warnings, ...)
- Cross-comparison and link creation

This is useful for:

- CCMS introduction and modularization of monolithic documents
- Preparation for Content Delivery of content with little or no metadata
- Merging technical documentation after an acquisition

Apart from efficiency benefits, enforcing metadata models and authoring rules by AI pushes the customer towards cleaning up grown structures.

Working into a complex field of knowledge the traditional way requires lots of reading and flipping over content that's irrelevant for the desired information.

The basic idea is similar to RAG, but in a broader scope.



- In the orange box, the user tells the process what to do research about, provides files and URLs as the base and optionally a style sample (e.g., some text from a scientific paper, if the result should be this way).
- The system first extracts all information about the scope and its perimeter from the content and stores it as modules with some axes of keywords, then creates a TOC.
- From the modules created, it builds modules filling the TOC positions.
- Finally, for each result module, the knowledge modules are searched for similar content using vector search, and the output topic is rounded out and linguistically adjusted.

This is another well-known AI application, and there are numerous solutions, like CoPilot.

Since we explicitly control the background data archive and (using the API) can integrate with Integrated Development Environments (IDE) like VisualStudio (Code), IntelliJ and others, **i2ai / i2store** can deliver where other approaches can't.

- With an IDE integration, the AI processes have access to the last version of the code base.
- Multiple solutions (backend service and frontend GUI) can be managed at the same time.
- History can be used for extracting knowledge (approaches that worked well or not so...).
- Most models are good at coding Python, C# etc. directly from the training. Historic or exotic languages are weaker. **i2ai / i2store** can manage valid code samples as reference for generation.
- Likewise, in higher languages, coding styles (how to name variables) and patterns (how to issue a Web API call the same way in the entire app) can be defined and applied.

Two examples are: retro computer games in for C64 & Co., and legacy mainframe languages still in production use (like COBOL or PL/1)

