uxodocs

Uxopian AI / uxodocs

Author: Your name

Date: 2026-03-13

Contents

Getting Started / Quick Start: Your First AI Exchange in 5 Minutes

This guide gets you from zero to a working AI response as fast as possible using the Docker Starter Kit.

Prerequisites

- **Docker** and **Docker Compose** installed.
- An **OpenAI API Key** (or any supported LLM provider key).

Step 1: Download the Starter Kit

> **DOWNLOAD**
>
> **uxopian-ai_docker_example.zip**

Extract the archive. You should have the following structure:

```
.
├── config/
│   ├── application.yml
│   ├── goals.yml
│   ├── llm-clients-config.yml
│   ├── llm-clients-config.yml.example
│   ├── mcp-server.yml
│   ├── metrics.yml
│   ├── opensearch.yml
│   └── prompts.yml
├── gateway-application.yaml
└── uxopian-ai-stack.yml
```

Step 2: Configure Your API Key

The LLM configuration uses environment variables. The simplest approach is to set your API key in the `uxopian-ai-stack.yml` file by adding it to the `environment` section of the `uxopian-ai-standalone` service:

```
environment:
  - OPENAI_API_KEY=sk-YOUR_OPENAI_API_KEY_HERE
```

Alternatively, you can replace `${OPENAI_API_KEY:}` directly in `config/llm-clients-config.yml` with your key.

> **OTHER PROVIDERS**
>
> You can use any supported provider (Azure, Anthropic, Gemini, Mistral, Ollama...). See Configuration Files for the full reference.

Step 3: Start the Stack

Start all services:

```
docker-compose -f uxopian-ai-stack.yml up -d
```

> **DOCKER IMAGES**
>
> The compose file references images from `artifactory.arondor.cloud:5001/`. If your registry differs, update the `image:` fields in `uxopian-ai-stack.yml` accordingly.

Wait a few seconds for OpenSearch to initialize. You can check the health:

```
curl http://localhost:8085/actuator/health
```

> **PORT MAPPING**
>
> The starter kit maps the service to **port 8085** on the host (`8085:8080`). All API calls from your machine go to `localhost:8085`.

> **WHY NO GATEWAY?**
>
> The starter kit runs in **development mode** (`SPRING_PROFILES_ACTIVE=dev`), which lets you call the AI service directly without a BFF Gateway. The service automatically fills in default values for missing authentication headers. In production, a BFF Gateway sits in front of the service and handles authentication, role injection, and tenant isolation.

Step 4: Create a Prompt

Tell the AI what to do by creating your first prompt:

```
curl -X POST "http://localhost:8085/api/v1/admin/prompts" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: quickstart" \
  -H "X-User-Id: admin" \
  -H "X-User-Roles: admin" \
  -d '{
    "id": "helloAI",
    "role": "system",
    "content": "You are a helpful assistant. Answer concisely.",
    "defaultLlmProvider": "openai",
    "defaultLlmModel": "gpt-5.1"
  }'
```

> **NOTE**
>
> The starter kit already includes pre-defined prompts in `config/prompts.yml` (e.g., `summarizeDocumentText`, `translate`). You can use them directly or create new ones via the API.

Step 5: Create a Conversation and Send a Message

First, create a conversation:

```
curl -X POST "http://localhost:8085/api/v1/conversations" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: quickstart" \
  -H "X-User-Id: demo-user"
```

Copy the `id` from the response, then send your first message:

```
curl -X POST "http://localhost:8085/api/v1/requests?
conversation=YOUR_CONVERSATION_ID" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: quickstart" \
  -H "X-User-Id: demo-user" \
  -d '{
    "inputs": [
      {
        "role": "user",
        "content": [
          {
            "type": "text",
            "value": "What is the capital of France?"
          }
        ]
      }
    ]
  }'
```

You should receive a JSON response with the AI's answer.

What's Next?

You've just completed your first AI exchange with uxopian-ai. Here's where to go from here:

- **Understand the system**: Read Core Concepts to learn about Prompts, Goals, and Conversations.
- **Manage prompts**: Learn how to manage prompts and goals with the API and the admin UI.
- **Explore the Admin Panel**: See your usage stats in the Dashboard.
- **Go deeper**: Discover the Templating Engine to build dynamic, context-aware prompts.
- **Understand security**: Learn how the BFF Gateway handles authentication in production and why this starter kit can skip it.

Getting Started / 📚 Guide: Uxopian AI Service Deployment

This guide covers the installation process for the **Uxopian AI** backend service and its vector database.

📦 Backend Deployment (AI Service)

This section covers the installation of the AI engine and its vector database. Two scenarios are possible: **Docker** (Recommended via Starter Kit) or **Standalone** (Native Java).

🚀 Scenario A: Docker Deployment (Starter Kit)

The Starter Kit provides a ready-to-use stack containing the AI service and an OpenSearch node.

🔹 Step 1: Download and Structure

> 💡 **DOWNLOAD**
>
> **uxopian-ai_docker_example.zip**

Once extracted, you should have the following directory structure:

```
.
├── config
│   ├── application.yml             # Main Spring configuration
│   ├── goals.yml                   # AI Goals definition
│   ├── llm-clients-config.yml      # API Keys and Model selection
(OpenAI, Mistral, etc.)
│   ├── llm-clients-config.yml.example  # Example with all
providers
│   ├── mcp-server.yml              # Model Context Protocol config
│   ├── metrics.yml                 # Micrometer & Actuator config
│   ├── opensearch.yml              # Vector database connection
│   └── prompts.yml                 # Pre-defined prompts
├── gateway-application.yaml        # Gateway config (if used)
└── uxopian-ai-stack.yml            # The docker-compose file
```

🔹 Step 2: Pull Images

Pull the required images from the registry configured in your `uxopian-ai-stack.yml`:

```
docker pull artifactory.arondor.cloud:5001/uxopian-ai:2026.0.0-ft1-rc3
# Note: The OpenSearch image is public and will be pulled automatically by the compose file.
```

> **REGISTRY**
>
> The compose file uses `artifactory.arondor.cloud:5001/` by default. If your organization hosts images on a different registry (e.g., `docker.uxopian.com/preview/`), update the `image:` fields in `uxopian-ai-stack.yml` accordingly.

🔹 Step 3: Environment Variable Configuration

The `uxopian-ai-stack.yml` file orchestrates the containers. **Do not modify the YAML structure**, but you must adapt the environment variables of the `uxopian-ai-standalone` service to ensure network communication.

There are two distinct communication flows to configure:

1. Integration Communication (AI to Document Service)

The AI must contact the Document Service (e.g., ARender Service Broker) to read document text via the internal Docker network.

Variable	Description	Example (Internal Docker)
`OPENSEARCH_HOST`	OpenSearch container hostname.	`uxopian-ai-opensearch-node1`
`OPENSEARCH_PORT`	OpenSearch port.	`9200`

2. Client Access Configuration (Browser to AI)

The user interface (running in the user's browser) must contact the AI service.

Variable	Description	Example (Public)
`UXOPIAN_AI_PORT`	Internal listening port of the service.	`8080`

Variable	Description	Example (Public)
`APP_BASE_URL`	Public URL of the AI application (for callbacks).	`http://localhost:8085`
`SPRING_PROFILES_ACTIVE`	Configuration profile (`dev` disables strict security).	`dev`

ABOUT THE `dev` PROFILE AND THE BFF GATEWAY

The starter kit ships with `SPRING_PROFILES_ACTIVE=dev` and **no Gateway service**. This means you can call the AI service directly — missing `X-User-*` headers are filled in with defaults (`User-development` / `Tenant-development`).

In a **production** deployment, you should remove the `dev` profile and deploy the Uxopian Gateway (BFF) in front of the AI service. The Gateway authenticates users, extracts their identity from JWT/OAuth2/LDAP tokens, and injects the `X-User-TenantId`, `X-User-Id`, `X-User-Roles`, and `X-User-Token` headers. The Gateway service is also included in this compose file (commented out) — uncomment the `uxopian-ai-gateway` block, configure its `provider`, and remove the `dev` profile to switch to a secured setup.

Step 4: Start

```
docker-compose -f uxopian-ai-stack.yml up -d
```

☕ Scenario B: Manual Installation (ZIP / Java)

Use this method for deployment on a standard server (VM Linux/Windows) without Docker.

Prerequisites:

- **Java 21** Runtime Environment (JRE).
- **OpenSearch 2.x** installed and running on the network.

🔹 Step 1: Installation

> 💡 **DOWNLOAD**
>
> **ai-standalone-2026.0.0-ft1-rc3-complete-package.zip**
>
> *Contact your Uxopian representative for access to this package.*

Unzip the archive:

```
unzip ai-standalone-2026.0.0-ft1-rc3-complete-package.zip
cd ai-standalone
```

🔹 Step 2: Configuration

All files are located in the `config/` folder. You **must** edit them:

- **`opensearch.yml`**: Enter the IP and credentials of your external OpenSearch cluster.

- `llm-clients-config.yml`: Configure your LLM providers (Azure OpenAI, Mistral, etc.) and API keys.
- `application.yml`: General settings (ports, logs).

Documentation is available at: Configuration

🔹 Step 3: Execution

Run the Java service:

```
java -jar ai-standalone.jar
```

> ⓘ **PRODUCTION RECOMMENDATIONS**

- Use `JAVA_OPTS` to allocate enough memory (e.g., `-Xmx4g`).
- Place the service behind a Reverse Proxy (NGINX/Apache) to handle SSL. :::

🔹 Step 4: Create Your First Prompt

Once the backend is running, you need to define what the AI should do by creating prompts via the API.

See the Managing Prompts and Goals guide for detailed instructions and examples.

Getting Started / 📚 Complete Guide: Uxopian AI Deployment & ARender Integration

This guide covers the entire process for deploying the **Uxopian AI** solution and integrating it into the **ARender** user interface.

📦 Part 1: Backend Deployment (AI Service)

This section covers the installation of the AI engine and its vector database. Two scenarios are possible: **Docker** (Recommended via Starter Kit) or **Standalone** (Native Java).

🚀 Scenario A: Docker Deployment (Starter Kit)

The Starter Kit provides a ready-to-use stack containing the AI service, an OpenSearch node, and a basic ARender stack for testing.

🔹 Step 1: Download and Structure

> 💡 **DOWNLOAD**
>
> **uxopian-ai_docker_example_arender.zip**

Once extracted, you should have the following directory structure:

```
.
├── arender/
│   └── configurations/
│       ├── arender-custom-client.properties  # ARender AI host &
button config
│       ├── arender-plugins.xml               # Plugin loader
│       └── toppanel-arender-ai-configuration.xml  # AI button
definitions
├── config/
│   ├── application.yml                 # Main Spring configuration
│   ├── goals.yml                       # AI Goals definition
│   ├── llm-clients-config.yml          # API Keys and Model
selection (OpenAI, Mistral, etc.)
│   ├── llm-clients-config.yml.example  # Example with all
providers
│   ├── mcp-server.yml                  # Model Context Protocol
config
│   ├── metrics.yml                     # Micrometer & Actuator
config
│   ├── opensearch.yml                  # Vector database
connection
│   └── prompts.yml                     # Pre-defined prompts
├── gateway-application.yaml            # Gateway config (if used)
└── uxopian-ai-stack.yml                # The docker-compose file
```

🔹 Step 2: Pull Images

Pull the required images from the registry configured in your `uxopian-ai-stack.yml`:

```
docker pull artifactory.arondor.cloud:5001/uxopian-ai:2026.0.0-ft1-
rc3-full
# Note: The OpenSearch and ARender images will be pulled
automatically by the compose file.
```

> ⓘ **REGISTRY**
>
> The compose file uses `artifactory.arondor.cloud:5001/` by default. If your organization hosts images on a different registry (e.g., `docker.uxopian.com/preview/`), update the `image:` fields in `uxopian-ai-stack.yml` accordingly.

🔹 Step 3: Environment Variable Configuration

The `uxopian-ai-stack.yml` file orchestrates the containers. **Do not modify the YAML structure**, but you must adapt the environment variables of the `uxopian-ai-standalone` service to ensure network communication.

There are two distinct communication flows to configure:

1. Server-to-Server Communication (AI Backend to ARender)

The AI must contact the ARender *Service Broker* to read document text. This happens via the internal Docker network.

Variable	Description	Example (Internal Docker)
`RENDITION_BASE_URL`	Internal URL of the ARender Service Broker.	`http://dsb-service:8761`
`OPENSEARCH_HOST`	OpenSearch container hostname.	`uxopian-ai-opensearch-node1`
`OPENSEARCH_PORT`	OpenSearch port.	`9200`

2. Client-to-Server Communication (Browser to AI)

The ARender Interface (running in the user's browser) must contact the AI.

Variable	Description	Example (Public)
`UXOPIAN_AI_PORT`	Internal listening port of the service.	`8080`
`APP_BASE_URL`	Public URL of the AI application (for callbacks).	`http://localhost:8085`
`SPRING_PROFILES_ACTIVE`	Configuration profile (`dev` disables strict security).	`dev`

> **ABOUT THE `dev` PROFILE AND THE BFF GATEWAY**
>
> This starter kit ships with `SPRING_PROFILES_ACTIVE=dev` and **no Gateway service**. This lets you call the AI service directly — missing `X-User-*` headers are filled in with defaults. In a **production** deployment, deploy the Uxopian Gateway (BFF) in front of the AI service to handle authentication, role enforcement, and token propagation. The Gateway service is included in the compose file (commented out) — see the gateway block in `uxopian-ai-stack.yml`.

> ⚠️ **NOTE ON ARENDER UI**
>
> In the **ARender UI** container configuration, do not forget to set `UXOPIAN_AI_HOST` to the public URL of the AI (e.g., `http://localhost:8085` or `https://ai.my-domain.com`).

🔹 Step 4: Start

```
docker-compose -f uxopian-ai-stack.yml up -d
```

☕ Scenario B: Manual Installation (ZIP / Java)

Use this method for deployment on a standard server (VM Linux/Windows) without Docker.

Prerequisites:

- **Java 21** Runtime Environment (JRE).
- **OpenSearch 2.x** installed and running on the network.

🔹 Step 1: Installation

> **DOWNLOAD**
>
> **ai-standalone-2026.0.0-ft1-rc3-complete-package.zip**
>
> *Contact your Uxopian representative for access to this package.*

Unzip the archive:

```
unzip ai-standalone-2026.0.0-ft1-rc3-complete-package.zip
cd ai-standalone
```

🔹 Step 2: Configuration

All files are located in the `config/` folder. You **must** edit them:

- **`opensearch.yml`**: Enter the IP and credentials of your external OpenSearch cluster.
- **`llm-clients-config.yml`**: Configure your LLM providers (Azure OpenAI, Mistral, etc.) and API keys.
- **`application.yml`**: General settings (ports, logs).

Documentation is available at: Configuration

🔹 Step 3: Execution

Run the Java service:

```
java -jar ai-standalone.jar
```

PRODUCTION RECOMMENDATIONS

- Use `JAVA_OPTS` to allocate enough memory (e.g., `-Xmx4g`).
- Place the service behind a Reverse Proxy (NGINX/Apache) to handle SSL.

🖥️ Part 2: ARender Configuration (Frontend)

This section details how to modify the ARender configuration to display AI buttons and interact with the deployed backend.

📂 Configuration File Structure

Whether using Docker or manual installation, prepare the following files according to this structure:

```
.
├── configurations
│   ├── arender-custom-client.properties      # Activates scripts
and configures AI host
│   ├── arender-plugins.xml                   # Imports Spring
beans
│   └── toppanel-arender-ai-configuration.xml # Defines the button
and JS action
└── public
    ├── web-components.css                    # Uxopian component
styles
    └── web-components.js                     # Uxopian component
JS logic
```

🔹 Step 1: Prompt Creation (Backend)

Before adding the button, the AI must know what to do. Create a prompt via the API.

See the Managing Prompts and Goals guide for detailed instructions. For this ARender integration, create a prompt with ID `summarizeDocMd` that uses `[[${documentService.extractTextualContent(documentId)}]]` to extract document content.

🔹 Step 2: Property Configuration (`arender-custom-client.properties`)

Edit `configurations/arender-custom-client.properties`. This file links the UI to the AI service.

```
# 1. Load CSS (ARender Style + AI Style)
style.sheet=css/arender-style.css,web-components.css

# 2. Load Web Component script at startup
arenderjs.startupScript=web-components.js

# 3. Add 'aiMenu' to the top toolbar (middle section)
topPanel.section.middle.buttons.beanNames=addStickyNoteAnnotationButt

# 4. Configure Public AI URL
# This is the address the user's browser will call
uxopian.ai.host=http://localhost:8085
# Production example: [https://ai.my-company.com](https://ai.my-compa

# 5. (Optional) Disable visual logs (toasters)
toaster.log.info.enabled=false
```

🔷 Step 3: Register Plugin (`arender-plugins.xml`)

Edit `configurations/arender-plugins.xml` to import your button configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-lazy-init="true" default-autowire="no"
    xmlns="[http://www.springframework.org/schema/beans]
(http://www.springframework.org/schema/beans)"
    xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance]
(http://www.w3.org/2001/XMLSchema-instance)"
    xsi:schemaLocation="
[http://www.springframework.org/schema/beans]
(http://www.springframework.org/schema/beans)
        [http://www.springframework.org/schema/beans/spring-
beans.xsd](http://www.springframework.org/schema/beans/spring-
beans.xsd)">

    <import resource="plume.xml"/>
    <import resource="html-plugin.xml"/>

    <import resource="toppanel-arender-ai-configuration.xml"/>

</beans>
```

🔷 Step 4: Button Definition (`toppanel-arender-ai-configuration.xml`)

This XML file defines the dropdown menu and the button triggering the AI call. It contains injected JavaScript code (`$wnd.createChat`).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-lazy-init="true" default-autowire="no"
    xmlns="[http://www.springframework.org/schema/beans]
(http://www.springframework.org/schema/beans)"
    xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance]
(http://www.w3.org/2001/XMLSchema-instance)"
    xsi:schemaLocation="[http://www.springframework.org/schema/beans]
(http://www.springframework.org/schema/beans)
        [http://www.springframework.org/schema/beans/spring-beans.xsd
(http://www.springframework.org/schema/beans/spring-beans.xsd)">

    <bean id="aiMenu"

class="com.arondor.viewer.client.toppanel.presenter.SubMenuButtonPres
       <constructor-arg value="aiMenu" />
       <constructor-arg value="AI" />
       <constructor-arg value="standardButton fas fa-robot toppanelBu
       <property name="enabled" value="true" />
       <property name="visibilityForTopPanel">
          <ref bean="topPanelVisibilityMode" />
       </property>
       <property name="orderedNamedList" value="summarizeDocMdButton"
    </bean>

    <bean id="summarizeDocMdButton"

class="com.arondor.viewer.client.toppanel.presenter.DropdownMenuItemP
       <constructor-arg value="summarizeDocMdButton"/>
       <constructor-arg value="Summarize Document"/>
       <constructor-arg value="standardButton fas fa-list toppanelBut
       <property name="enabled" value="true" />
       <property name="closingOnClick" value="true" />
       <property name="buttonHandler">
          <bean
class="com.arondor.viewer.client.jsapi.toppanel.JSCallButtonHandler">
             <property name="jsCode">
                <value>
```

```
try {
    // Call the global function exposed by web-components.js
    $wnd.createChat({
        endpoint: "${uxopian.ai.host}",   // Variable injected from .
        wsEndpoint: "${uxopian.ai.host}", // Variable injected from .
        request: {
            inputs: [{
                role: 'user',
                content: [{
                    type: 'PROMPT',
                    value: 'summarizeDocMd', // Prompt ID defined in
                    payload: {
                        // Get current document ID via ARender JS API
                        documentId: $wnd.getARenderJS().getCurrentDoc
                    }
                }]
            }]
        }
    });
} catch(e) {
  console.log('Error launching AI Chat: ' + e);
}
                </value>
              </property>
          </bean>
       </property>
    </bean>
</beans>
```

🛠️ Part 3: Applying Changes (ARender Deployment)

Once your configuration files are ready, apply them to your ARender instance.

🅰️ Option A: Docker Integration (Custom Image Build)

If using Docker for ARender, you **must** build a new image containing these configurations. Volume mounting alone can sometimes cause permission issues or file overwrites.

1. Create Dockerfile At the root of your folder containing `configurations/` and `public/`:

```
# Stage 1: File Preparation
FROM alpine as builder
WORKDIR /app
COPY configurations/ configurations/
COPY public/ public/

# Stage 2: Final ARender Image
FROM artifactory.arondor.cloud:5001/arender-ui-springboot:2023.16.0

# Copy XML/Properties configurations
COPY --from=builder /app/configurations/*
/home/arender/configurations/

# Copy Web resources (JS/CSS)
COPY --from=builder /app/public/* /home/arender/public/
```

2. Build Image

```
docker build -t my-company/arender-ui-ai:custom .
```

3. Update docker-compose In your `docker-compose.yml`, replace the `ui` service image with `my-company/arender-ui-ai:custom`.

🅱️ Option B: Manual Installation (Server)

For a standard installation (Tomcat or Executable Jar):

1. **Configurations:** Copy the contents of your `configurations/` folder (the 3 files) to your ARender installation's config folder (`$ARENDER_HOME/configurations/`).
2. **Web Resources:** Copy `web-components.js` and `web-components.css` to your installation's public folder (`$ARENDER_HOME/public/`).
3. **Restart:** Restart the ARender service to load the new Spring Beans.

Understanding Uxopian-ai / Architecture Overview

This section provides insight into the framework's design, covering both the high-level components and the software-level interactions.

Component Architecture

This section provides insight into the framework's design, covering both the high-level components and the software-level interactions, specifically highlighting the security integration via the BFF pattern.

The **uxopian-ai** framework is designed as a backend microservice that sits behind a security gateway. It is composed of several key components working in concert.

Component Descriptions

Client Application User-facing application (e.g., **ARender**, **FlowerDocs**) that initiates requests. It never communicates directly with **uxopian-ai**.

BFF / Gateway (Security Layer) Entry point for all traffic. The Gateway is a **standalone, independently deployed service** built on **Spring Cloud Gateway** (reactive) with a pluggable auth provider system. Responsible for:

- **Authenticating** the user via a pluggable provider (OAuth2, JWT, LDAP, or the `DevProvider` for development)
- **Enriching** requests with identity headers (`X-User-TenantId`, `X-User-Id`, `X-User-Roles`, `X-User-Token`)
- **Enforcing** role-based access control (e.g., admin-only paths)
- **Proxying** the request to the backend service

The Gateway processes each request through a filter pipeline: `DefaultProviderHeaderFilter` (route matching) → `AuthFilter` (authentication + header injection) → Spring Cloud Gateway (routing). See Security Model for the full pipeline details and provider reference.

uxopian-ai Service The core of the framework. This standalone Java application:

- Exposes the REST API (consumed by the BFF)
- Manages conversations and messages per Tenant ID
- Resolves Goals and Prompts via the templating engine
- Connects to external LLM providers using the `llm-clients` module

OpenSearch Primary data store for:

- Conversations
- Messages
- Prompts
- Goals

ARender Rendition Service External service used to fetch document content (e.g., extracting text).

Qdrant Optional vector database enabling RAG (Retrieval-Augmented Generation).

External LLM Providers Third-party services (OpenAI, Azure, etc.) handling natural language processing.

Software Architecture (Request Flow)

To understand how the components interact, here is the lifecycle of a typical API call: sending a message that triggers a Goal.

The request flow emphasizes the role of the **BFF** in establishing the security context before the service logic executes.

Sequence Diagram: Executing a Goal

Workflow Steps

1. **Client Request** Client sends a request to the BFF (e.g., `POST /api/v1/requests`) with a goal input such as `"compare"`.
2. **Authentication & Injection** The BFF authenticates, then injects `X-User-TenantId`, `X-User-Id`, etc.
3. **Context Establishment** uxopian-ai reads the headers to derive the security context.
4. **Goal Resolution** The service queries OpenSearch for Goals matching `"compare"` in the tenant.
5. **Filter Evaluation** SpEL filters narrow the choice to a specific `promptId` (e.g., `"detailedComparison"`).

6. **Prompt & Context Retrieval** The Prompt definition and conversation history are loaded.
7. **Template Rendering** Thymeleaf produces the final LLM prompt, optionally pulling external content.
8. **LLM Interaction** The prompt is sent to the configured LLM provider.
9. **Persistence** User message and LLM response are saved in OpenSearch.
10. **Response** The final response is returned through the BFF to the client.

Understanding Uxopian-ai / 🧠 Core Concepts

This section explains the primary entities and concepts that form the **uxopian-ai** framework. Understanding these concepts is essential for effective configuration and interaction.

🔐 Multi-Tenancy & Users

uxopian-ai is built with a multi-tenant architecture from the ground up, allowing for secure and logical separation of data within a single deployment.

- **Tenants**: Every interaction is scoped to a specific tenantId . This ensures that conversations, stats, and configurations are isolated per tenant.
- **Users**: Users are identified by an ID and assigned specific roles (including an admin flag) . This role-based access control governs access to the Admin API and sensitive operations.

🔌 Providers & Models

A **Provider** is a connector to an external Large Language Model (LLM) service. The framework uses providers to abstract the specific implementation details of each LLM service.

- **Providers**: Providers are configured dynamically per tenant via `LlmProviderConf` entities stored in OpenSearch. You can manage them

through the Admin API or the Admin UI. The list of registered provider types (e.g., `openai`, `azure`, `anthropic`) is available via `GET /api/v1/admin/llm/providers`.

- **Models**: Each provider configuration includes a list of model configurations (`LlmModelConf`), each with an alias (`llmModelConfName`) and the actual model identifier (`modelName`). The framework tracks model capabilities, specifically whether a model supports **multi-modal** inputs or **function calling**.

Choosing the Right Model

Not all models are equal. Choosing the right one for each prompt is critical for balancing **response quality**, **speed**, and **cost**.

Model Tier	Examples	Strengths	Best For
Flagship	`gpt-5.1`, `gpt-5`, `claude-3-opus`, `gemini-2.5-pro`	Deepest reasoning, highest accuracy	Complex analysis, multi-step logic, detailed comparisons
Balanced	`gpt-4.1`, `gpt-4o`, `claude-3-5-sonnet`, `gemini-2.5-flash`	Good reasoning with faster response	General-purpose use, summaries, document Q&A
Fast & Cheap	`gpt-5-mini`, `gpt-4.1-mini`, `gpt-4o-mini`, `gemini-2.5-flash-lite`	Low latency, low cost per token	High-volume tasks, simple extractions, intermediate map-reduce steps

Model Tier	Examples	Strengths	Best For
Ultra-light	`gpt-5-nano`, `gpt-4.1-nano`, `gpt-3.5-turbo`, `gemini-2.0-flash-lite`	Minimal cost, fastest response	Classification, keyword extraction, routing decisions
Reasoning	`o3-mini`, `o4-mini`	Extended chain-of-thought reasoning	Math, logic puzzles, code generation (no multi-modal)

PRACTICAL GUIDANCE

- **Start balanced, optimize later.** Use `gpt-5.1` or `gpt-4.1` to validate your prompt logic, then switch to a `mini` or `nano` variant once it works.
- **Use cheap models for intermediate steps.** In a Map-Reduce helper, the map phase processes many chunks — use a fast model like `gpt-5-mini` there, and reserve the flagship model for the final reduce step.
- **Check capability flags.** If your prompt uses images (`requiresMultiModalModel: true`) or tool calls (`requiresFunctionCallingModel: true`), verify that your chosen model supports them in its `LlmModelConf`. The `o3-mini`/`o4-mini` reasoning models, for example, do **not** support multi-modal inputs.
- **Monitor with the admin panel.** The Statistics dashboard shows token consumption and model distribution — use it to identify prompts that could benefit from a cheaper model.

Model Capabilities

Each model configuration (`LlmModelConf`) declares two capability flags:

- **`multiModalSupported`**: Can the model process image inputs (Base64 or URL)? Required for prompts that inject images via `documentService.getPageImage(...)`.
- **`functionCallSupported`**: Can the model invoke external tools (MCP, custom tools)? Required for prompts that trigger custom tools.

When a prompt is configured with `requiresMultiModalModel: true` or `requiresFunctionCallingModel: true`, the framework verifies that the selected model actually supports the required capability. If it does not, the request will fail with a clear error message.

⚖️ Parameter Precedence

When making a call to an LLM, you can specify configuration parameters at multiple levels. The framework uses a clear **5-level** order of precedence to determine which values to use:

1. **API Call Parameters:** Values passed directly in the request (e.g., `provider`, `model`, `temperature`) always take the highest priority.
2. **Prompt Defaults:** If a parameter is not specified in the API call, the framework looks for defaults defined in the Prompt configuration (`defaultLlmProvider`, `defaultLlmModel`).
3. **Provider Model Config** (`LlmModelConf`): Per-model settings defined in the provider configuration (e.g., a specific `temperature` or `maxTokens` for `gpt-5-mini`).
4. **Provider Global Config** (`LlmProviderConf.globalConf`): Shared settings defined at the provider level that apply to all models under that provider.

5. **YAML Global Defaults:** The fallback values defined in `llm.default.*` in `llm-clients-config.yml`.

> **EXAMPLE**
>
> You define a global default of `gpt-5.1` in `llm-clients-config.yml` with `temperature: 0.7`. Your OpenAI provider configuration sets `temperature: 0.5` globally and `temperature: 0.2` specifically for the `gpt5-mini` model. Your "quick classification" prompt overrides the model with `defaultLlmModel: gpt5-mini`. A specific API call can still override everything by passing `"model": "gpt-4.1"` in the request body. This five-level system lets you set sensible defaults while retaining full flexibility per provider, per model, per prompt, and per call.

💬 Conversations

A **Conversation** is a container that groups a sequence of exchanges between a user and the AI.

- **Persistence**: Conversations are persistent entities with a title, update timestamp, and a record of the last used LLM provider and model .
- **Context**: The framework automatically manages the context window, retrieving recent messages to ensure stateful interactions.
- **Lifecycle**: Conversations can be listed, retrieved by ID, or deleted via the API .

⇄ Requests & Inputs (Multi-Modal)

A **Request** represents a single exchange (a "turn") within a conversation. Unlike simple text messages, **uxopian-ai** requests are rich and multi-modal.

- **Structure**: A request consists of a list of inputs (what was sent) and an answer (what the AI generated) .
- **Multi-Modal Content**: The Input block supports various content types. It is not limited to text; it can handle image content or references to prompt IDs and goal names .
  - **Dynamic Images**: The image input type also supports **Thymeleaf templating**, allowing you to dynamically inject image data (e.g., `[[${documentService.getPageImage(documentId, 1, 800)}]]`).
  - **Constraint**: If using templating for images, the expression **must resolve to a raw Base64 string** representing the image.
- **Token Tracking**: Every request logs inputTokenCount and outputTokenCount for precise cost monitoring .
- **Feedback**: Each request can be tagged with user feedback (Good, Bad, or Neutral) to help tune performance over time .

📜 Prompts

A **Prompt** is a reusable, templated instruction sent to a model. Prompts are central to standardizing AI behavior.

- **Templating**: Prompts use a templating engine (Thymeleaf) allowing for dynamic variable injection.
- **Configuration**: Beyond the text content, a prompt stores configuration flags like reasoningDisabled, requiresMultiModalModel, and requiresFunctionCallingModel .
- **ROI Estimation**: Prompts include a timeSaved attribute (in seconds). This is used to calculate the Return on Investment (ROI) by estimating how

much human time is saved every time the prompt is used .

🎯 Goals

A **Goal** is a high-level orchestration unit. It decouples the user's intent from the specific prompt used, allowing for dynamic selection based on context.

- **Logic**: A goal maps a goalName to a specific promptId based on a filter expression .
- **Filtering**: The filter typically uses Spring Expression Language (SpEL) to evaluate the payload (e.g., `[[${documentType == 'contract'}]]`).
- **Priority**: Goals have an index property, allowing you to define priority order when multiple goals might match a scenario .

📊 Analytics & Statistics

uxopian-ai includes a comprehensive analytics engine accessible via the Admin API.

- **Global Stats**: Tracks total conversations, requests, tokens, and aggregated time saved .
- **Time Series**: Provides activity data (requests, tokens, time saved) aggregated by time intervals (e.g., DAY) to visualize trends .
- **Adoption & ROI**:
  - **Feature Adoption**: Tracks the usage rate of advanced features like multi-modal and function calling .
  - **Time Saved**: Ranks prompts by the total estimated hours they have saved users .

Understanding Uxopian-ai / The Templating Engine

Uxopian-ai uses **Thymeleaf** combined with **Spring Expression Language (SpEL)** to make prompts dynamic. Instead of sending static text to the LLM, you can inject runtime data, call Java services, and build conditional logic directly inside your prompt definitions.

Basic Syntax

All dynamic expressions use the Thymeleaf inline text output syntax:

```
[[${expression}]]
```

Everything inside `[[${...}]]` is evaluated at render time before the prompt is sent to the LLM.

Variable Resolution

When a request is sent to uxopian-ai with a **payload**, each key in the payload JSON becomes a **top-level template variable**. There is no `payload.` prefix.

Example: Given this API request payload:

```
{
  "documentId": "doc-abc-123",
  "language": "french"
}
```

The prompt template can reference these variables directly:

```
Translate the following document into [[${language}]]:

[[${documentService.extractTextualContent(documentId)}]]
```

At render time, `language` resolves to `"french"` and `documentId` resolves to `"doc-abc-123"`.

Java Services (Helpers)

Registered Spring beans are also available as top-level variables in templates. The built-in services include:

- **`documentService`** — Extract text or images from documents (e.g., via ARender or FlowerDocs).
- **`promptService`** — Render other prompts or access prompt definitions.

You call their public methods directly:

```
[[${documentService.extractTextualContent(documentId)}]]
```

You can also create your own custom helpers. See Creating Custom Helpers for details.

Conditional Logic

Use SpEL ternary expressions for conditional rendering:

```
Translate the following document in [[${language != null ? language
: 'english'}]]:

[[${documentService.extractTextualContent(documentId)}]]
```

If `language` is present in the payload, it is used; otherwise, it defaults to `"english"`.

Iteration

Use Thymeleaf's `th:each` to iterate over lists from the payload. This is useful when processing multiple documents in a single request.

Payload:

```
{
  "documentIds": ["doc-001", "doc-002", "doc-003"]
}
```

Prompt template:

```
Please provide a detailed, point-by-point comparison of the
following documents:

[# th:each="docId, iterStat : ${documentIds}"]
Document [[${iterStat.count}]]:
[[${documentService.extractTextualContent(docId)}]]
[/]
```

The `[# th:each="..."]...[/]` block repeats for each item in the list.

Composition (Prompt-in-Prompt)

A prompt can include the rendered output of another prompt using `promptService.renderPrompt()`. This lets you reuse common formatting instructions or personas across multiple prompts.

```
Summarize the following document.
[[${promptService.renderPrompt('markdownResponse')}]]

Document content:
[[${documentService.extractTextualContent(documentId)}]]
```

Here, `markdownResponse` is a separate prompt that contains formatting rules (e.g., "Use Markdown headers and bullet points"). It is rendered inline before the final prompt is sent to the LLM.

Conversation History

The variable `messages` gives access to the current conversation history. You can inject it into a prompt to provide the LLM with prior context:

```
Here is the conversation so far:
[[${messages}]]

Now answer the following question:
[[${userQuestion}]]
```

Going Further

- Creating Custom Helpers — Expose your own Java services in templates.
- Creating Advanced Helpers — Implement Map-Reduce patterns for large documents.
- Managing Prompts and Goals — CRUD operations on prompts via the API.

Understanding Uxopian-ai / Security Model (BFF Pattern)

Uxopian-ai is a backend microservice designed to run behind a **BFF (Backend for Frontend)** or **API Gateway**. It should **never** be exposed directly to the public internet. This page explains why, how the Gateway works internally, and how to work without it during development.

Why a BFF Gateway?

The uxopian-ai service does **not** handle authentication itself — by design. This separation provides several advantages:

- **Single responsibility**: The AI service focuses on prompt resolution and LLM orchestration. The Gateway focuses on security.
- **Pluggable authentication**: Your organization can use OAuth2, JWT, SAML, LDAP, or any custom mechanism. The AI service does not need to change.
- **Token propagation**: The Gateway forwards the original user token (`X-User-Token`), which Helpers and integrations can use to call other APIs (FlowerDocs, ARender) on behalf of the authenticated user.
- **Role-based access control**: The Gateway maps roles from your identity provider into the `X-User-Roles` header. The AI service uses this to gate admin operations.
- **Tenant isolation**: The Gateway extracts the organization/scope from the authenticated session and injects it as `X-User-TenantId`, enabling full

multi-tenancy.

In a typical deployment, three layers are involved:

1. **Client Application** — The user-facing app (ARender, FlowerDocs, a custom web app).
2. **BFF / Gateway** — Authenticates the user and injects identity headers.
3. **Uxopian-ai Service** — Receives pre-authenticated requests and processes them.

The Gateway is the **only** component exposed to the network.

The Uxopian Gateway

The Uxopian Gateway is a **standalone service**, built and deployed independently from the AI service. It is built on **Spring Cloud Gateway** (reactive) and sits in front of the AI service, handling authentication via a **pluggable provider** system. The built-in auth providers (DevProvider, FlowerDocsProvider, Fast2Provider) are part of the Gateway service.

Request Processing Pipeline

Every request passes through two filters before reaching the backend:

1. **`DefaultProviderHeaderFilter`** — Uses `AntPathMatcher` to match the incoming request URL against the configured routes. When a match is found, it injects the `X-Provider-ID` internal header with the provider name from the route config (e.g., `FlowerDocsProvider`, `Fast2Provider`, `DevProvider`).

2. **`AuthFilter`** — Reads the `X-Provider-ID` header, loads the corresponding `AuthProvider` implementation, and calls its `authenticate(request)` method. The provider returns an `AuthenticatedUser` with `id`, `roles`, `tenantId`, and `token`. The filter then **enriches** the downstream request with these headers:

Enriched Header	Source
`X-User-Id`	`AuthenticatedUser.id`
`X-User-Roles`	`AuthenticatedUser.roles` (comma-joined)
`X-User-TenantId`	`AuthenticatedUser.tenantId`
`X-User-Token`	`AuthenticatedUser.token`

   The filter also creates a Spring Security `UsernamePasswordAuthenticationToken` and places it in the reactive `SecurityContext`, enabling role-based path security rules.

3. **Spring Cloud Gateway** — Forwards the enriched request to the backend URI (`http://ai-standalone-service:8080`).

Pluggable Auth Providers

The Gateway uses a **plugin architecture**. Auth providers are loaded dynamically from JAR files at startup:

- At boot, the Gateway scans the `provider/` directory (configurable via `auth.provider.path`) for provider implementations.
- Each provider is a Spring `@Service` with a name (e.g., `@Service("FlowerDocsProvider")`).

- The route configuration references that name in the `provider:` field.

Built-in providers:

Provider	Authentication Method
`DevProvider`	Reads identity from raw request headers (for development)
`FlowerDocsProvider`	Validates FlowerDocs JWT tokens (from `token` header or `SESSION` cookie)
`Fast2Provider`	Decrypts Fast2 JWT tokens via a remote public key

To integrate with your organization's identity system (Keycloak, Azure AD, custom SSO), you implement the `AuthProvider` interface and drop the JAR into the `provider/` directory. See Adding a Custom Auth Provider.

Gateway Route Configuration

Routes are defined in the Gateway's `application.yml`:

```yaml
app:
  gateway:
    provider-header: X-Provider-ID
  routes:
    - id: uxopian-ai
      uri: http://ai-standalone-service:8080
      prefix: /gui/gateway/uxopian-ai/
      path: /gui/gateway/uxopian-ai/**
      provider: FlowerDocsProvider        # ← Which auth provider
to use
      security:
        - path: /.well-known/**
          public: true                     # ← No auth required
        - path: /swagger-ui/**
          public: true
        - path: /prompt/**
          roles: [ "ADMIN" ]              # ← Requires ADMIN role
        - path: /goal/**
          roles: [ "ADMIN" ]
```

Each route maps a URL pattern to a backend service and specifies which auth provider to use and what security rules apply per path.

Authentication Headers

When deploying in production, your Gateway (Uxopian's or your own) must inject the following headers into every request before it reaches uxopian-ai:

Header	Description	Required
`X-User-TenantId`	Isolates data per tenant (organization).	**Yes**
`X-User-Id`	Unique identifier for the user.	**Yes**
`X-User-Roles`	Comma-separated list of roles (e.g., `admin,user`).	No
`X-User-Token`	Original user token (if needed for downstream context).	No

The `X-User-Roles` header controls access to admin endpoints. Operations under `/api/v1/admin/*` require the `admin` role. The `X-User-Token` is propagated to Helpers and integrations that need to call external APIs (e.g., FlowerDocs, ARender) on behalf of the user.

Production Mode

In production, the Gateway intercepts every incoming request, validates the user's credentials, and injects the `X-User-*` headers. The uxopian-ai service trusts these headers implicitly.

Flow:

1. User sends a request to the Gateway.
2. Gateway selects the appropriate `AuthProvider` based on the route.
3. Provider authenticates (OAuth2, JWT, SAML...) and returns an `AuthenticatedUser`.

4. Gateway enriches the request with `X-User-TenantId`, `X-User-Id`, `X-User-Roles`, `X-User-Token`.
5. Gateway forwards the request to uxopian-ai.
6. Uxopian-ai reads the headers and establishes the security context (`AiContext`).
7. All data access (conversations, prompts, stats) is scoped to the tenant.

> ⚠ **NEVER EXPOSE UXOPIAN-AI DIRECTLY**
>
> Without a Gateway, anyone can forge `X-User-*` headers and impersonate any user or tenant. The Gateway is what makes these headers trustworthy.

Development Mode

For local development and testing, there are **two levels** of dev-mode that work together:

1. The `DevProvider` (Gateway Side)

The `DevProvider` is a built-in auth provider that trusts raw request headers without any validation. Instead of decrypting a JWT or calling an OAuth endpoint, it simply reads `X-User-Id`, `X-User-Roles`, and `X-User-Tenant` directly from the incoming request.

```
@Service("DevProvider")
public class DevProvider implements AuthProvider {
    @Override
    public Mono<AuthenticatedUser> authenticate(ServerHttpRequest
request) {
        String userId = request.getHeaders().getFirst("X-User-Id");
        String rolesRaw = request.getHeaders().getFirst("X-User-
Roles");
        String tenantId = request.getHeaders().getFirst("X-User-
Tenant");
        // ... builds AuthenticatedUser from these headers
    }
}
```

To activate it, configure a route with `provider: DevProvider` in the Gateway's `application.yml`. This lets you call the Gateway with plain headers (e.g., from `curl` or Postman) without needing real credentials.

2. The `dev` Profile (AI Service Side)

When the uxopian-ai service is started with `SPRING_PROFILES_ACTIVE=dev`, its internal `AuthFilter` injects default credentials if the `X-User-*` headers are missing:

- **Default Tenant:** `Tenant-development`
- **Default User:** `User-development`

This means you can call the AI service **directly** (bypassing the Gateway entirely) and it will still work:

```
# No X-User-* headers needed — dev profile fills in defaults
curl http://localhost:8085/api/v1/conversations
```

You can also override specific headers while letting others fall back to defaults:

```
# Custom tenant, default user
curl -H "X-User-TenantId: my-test-tenant"
http://localhost:8085/api/v1/conversations
```

STARTER KIT USES DEV MODE

The Docker Starter Kit ships with `SPRING_PROFILES_ACTIVE=dev` and no Gateway service. This is why the Quick Start curl commands work directly against `localhost:8085` — the dev profile handles the missing headers automatically.

NEVER USE `dev` IN PRODUCTION

The `dev` profile **disables all authentication enforcement**. Any request without headers will be accepted as `User-development` in `Tenant-development`. Combined with no Gateway, this means zero access control.

When to Use What

Scenario	Gateway	AI Service Profile	How Auth Works
Production	Uxopian Gateway + real provider (e.g., `FlowerDocsProvider`)	Default (no `dev`)	Gateway validates credentials, injects headers. AI service rejects

Scenario	Gateway	AI Service Profile	How Auth Works
			requests without headers.
Staging / Integration	Uxopian Gateway + `DevProvider`	Default (no `dev`)	You pass raw headers through the Gateway. AI service still requires headers.
Local development	None	`dev`	Call the AI service directly. Missing headers get default values.

Multi-Tenancy

Every piece of data in uxopian-ai is scoped to a **Tenant ID**:

- **Conversations** belong to a specific tenant.
- **Prompts and Goals** can be global or tenant-specific.
- **Statistics** are aggregated per tenant.

The `X-User-TenantId` header determines which tenant's data a request can access. A user from `tenant-A` cannot see conversations from `tenant-B`, even if they share the same uxopian-ai instance.

This logical separation is enforced at the data layer (OpenSearch queries always include the tenant filter).

Going Further

- Adding a Custom Auth Provider — Implement your own authentication logic for the Gateway.
- Architecture Overview — See the full component and sequence diagrams.
- Configuration Files — Gateway and AI service configuration reference.

How-To Guides / Managing Prompts and Goals

This guide explains how to create, update, and manage Prompts and Goals using the REST API and the admin interface.

For a visual guide on managing these resources via the UI, refer to the Admin Interface Guide.

For a deep dive into the templating syntax used in prompt content, see The Templating Engine.

Managing Prompts and Goals via the API

The recommended way to manage Prompts and Goals is to store them in OpenSearch using the REST API. This allows for dynamic updates without restarting the service.

> ⚠ **SECURITY NOTE**
>
> These are **Admin** operations. Your Gateway must inject the `X-User-Roles: admin` header for these requests to succeed.

API Operations

Refer to the Swagger documentation for the complete schema details.

Prompts

- **Save a Prompt**: `POST /api/v1/admin/prompts`
- **Update a Prompt**: `PUT /api/v1/admin/prompts`
- **Get a Specific Prompt**: `GET /api/v1/admin/prompts/{id}`
- **Delete a Prompt**: `DELETE /api/v1/admin/prompts/{id}`

Goals

- **Save a Goal**: `POST /api/v1/admin/goals`
- **Update a Goal**: `PUT /api/v1/admin/goals`
- **Get All Goals**: `GET /api/v1/admin/goals`
- **Delete a Goal**: `DELETE /api/v1/admin/goals/{id}`

Example: Creating a New Prompt

Use this endpoint to store a prompt configuration, including its default LLM settings.

cURL Request

```
curl -X POST "http://localhost:8080/api/v1/admin/prompts" \
-H "Content-Type: application/json" \
-H "X-User-TenantId: enterprise-corp-a" \
-H "X-User-Id: admin-user" \
-H "X-User-Roles: admin" \
-d '{
  "id": "summarizeDocumentText",
  "role": "user",
  "content": "Summarize the following document in a plain text
format:\n\n[[${documentService.extractTextualContent(documentId)}]]",
  "defaultLlmProvider": "openai",
  "defaultLlmModel": "gpt-5.1",
  "timeSaved": 60
}'
```

Note: The `timeSaved` field (in seconds) is used to calculate ROI stats in the admin panel.

> **CHOOSING THE RIGHT MODEL**
>
> The `defaultLlmModel` you assign to a prompt has a direct impact on **response quality**, **speed**, and **cost**. Use a flagship model (`gpt-5.1`, `gpt-5`) for complex analysis, a balanced model (`gpt-4.1`, `gpt-4o`) for general use, or a fast model (`gpt-5-mini`, `gpt-4.1-nano`) for high-volume, simple tasks. See Choosing the Right Model for a complete guide.

Example: Creating a New Goal

A Goal maps a specific context to a prompt ID.

cURL Request

```
curl -X POST "http://localhost:8080/api/v1/admin/goals" \
-H "Content-Type: application/json" \
-H "X-User-TenantId: enterprise-corp-a" \
-H "X-User-Id: admin-user" \
-H "X-User-Roles: admin" \
-d '{
  "goalName": "compare",
  "promptId": "detailedComparison",
  "filter": "[[${documentType == '\''contract'\''}]]",
  "index": 125
}'
```

Examples of Prompt and Goal Definitions

Here are practical examples of how to structure your Goal and Prompt logic.

1. Goal Logic (Orchestration)

Goals use the `index` property to determine priority (lower numbers are checked first) and a `filter` to match the context.

Logic:

1. Check if `documentType` is 'contract'. If yes, use `detailedComparison`.
2. Otherwise, fall back to `genericComparison`.

```
[
  {
    "goalName": "compare",
    "promptId": "detailedComparison",
    "filter": "[[${documentType == 'contract'}]]",
    "index": 125
  },
  {
    "goalName": "compare",
    "promptId": "genericComparison",
    "filter": "true",
    "index": 1000
  }
]
```

2. Prompt: Conditional Logic

This prompt uses a SpEL expression to dynamically set the target language.

```
Translate the following document in [[${language != null} ?
${language} : 'english']]:

[[${documentService.extractTextualContent(documentId)}]]
```

3. Prompt: Iteration

This prompt iterates over a list of document IDs from the payload to compare multiple documents within a single request.

```
Please be exhaustive and provide a very detailed, point-by-point
comparison.
Compare the following documents:

[# th:each="docId, iterStat : ${documentIds}"]
Document content [[${iterStat.count}]] :
[[${documentService.extractTextualContent(docId)}]]
[/]
```

4. Prompt: Composition (Prompt-in-Prompt)

A prompt can call another prompt. Here, `summarizeDocumentMarkdown` reuses the formatting rules defined in a separate prompt named `markdownResponse`.

```
Summarize the following document.
[[${promptService.renderPrompt('markdownResponse')}]]

Document content:
[[${documentService.extractTextualContent(documentId)}]]
```

5. Prompt: System Persona

A generic `basePrompt` can be used to define the persona and core instructions for the AI.

```
You are Nono. You were born in 2025.
Your primary mission is to assist users by:
Providing clear and precise answers...
```

Web Interface for Prompt Management

In addition to the REST API, **uxopian-ai** includes a built-in web interface that lets you visually manage prompts.

> Access: `https://<your-uxopian-endpoint>/ai`

Through this interface, you can:

- View and search existing Prompts.
- Edit their fields (Content, Filters, Model Settings).
- Add new Prompts.
- Delete or reorder items interactively.

For a full walkthrough of the interface, see the Admin Interface Guide.

How-To Guides / Using the REST API

This guide provides practical examples for the most common tasks in **uxopian-ai**: managing conversations, sending requests, orchestrating goals, and accessing admin endpoints.

For details on the security model and authentication headers, see Security Model (BFF Pattern).

Prerequisites: Authentication Headers

Uxopian-ai never handles authentication itself — it relies on a **BFF Gateway** to validate credentials and inject identity headers into every request. See Security Model (BFF Pattern) for the full architecture.

Header	Description	Required
`X-User-TenantId`	Isolates data per tenant (organization).	**Yes**
`X-User-Id`	Unique identifier for the user.	**Yes**
`X-User-Roles`	Comma-separated list of roles (e.g., `admin,user`).	No

Header	Description	Required
`X-User-Token`	Original user token, forwarded to integrations (FlowerDocs, ARender).	No

The Gateway authenticates the user (OAuth2, JWT, LDAP...), extracts identity from the session, and enriches the request with these `X-User-*` headers before forwarding it to uxopian-ai. The AI service trusts these headers implicitly — which is why uxopian-ai must **never** be exposed directly to the network.

> **DEVELOPMENT MODE**
>
> In development mode (`SPRING_PROFILES_ACTIVE=dev`), the AI service accepts requests without headers and fills in defaults (`User-development` / `Tenant-development`). The examples below include explicit headers so they work in both dev and production environments. See Development Mode for details.

---

1. Creating a Conversation

Conversations are the container for all history and context. They are scoped to the specific `X-User-TenantId` provided in the headers.

- **Endpoint:** `POST /api/v1/conversations`

Example: Gateway forwarding a creation request

cURL

```
curl -X POST "http://localhost:8080/api/v1/conversations" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: user-james-bond" \
  -H "X-User-Roles: user"
```

JavaScript (Node.js / Gateway Logic)

```
const createConversation = async (tenantId, userId) => {
  const response = await fetch(
    "http://uxopian-ai-service:8080/api/v1/conversations",
    {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        "X-User-TenantId": tenantId,
        "X-User-Id": userId,
      },
    }
  );

  const data = await response.json();
  console.log("New Conversation ID:", data.id);
  return data;
};
```

---

2. Sending a Text Request (Non-Streaming)

Send a prompt to the LLM within a conversation. The `X-User-TenantId` ensures the user only accesses conversations they belong to.

- **Endpoint:** `POST /api/v1/requests`
- **Query Parameter:** `conversation` (Required)

Example: Standard User Query

cURL

```
curl -X POST "http://localhost:8080/api/v1/requests?
conversation=123e4567-e89b-12d3-a456-426614174000" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: user-james-bond" \
  -d '{
    "inputs": [
      {
        "role": "user",
        "content": [
          {
            "type": "text",
            "value": "How do I reset my password?"
          }
        ]
      }
    ]
  }'
```

---

3. Triggering a Goal (Orchestration)

Use the `goal` type to let **uxopian-ai** select the correct prompt based on the payload context.

- **Content Type:** `goal`
- **Payload:** Context data for the filter engine.

Example: Intelligent Goal Routing

cURL

```
curl -X POST "http://localhost:8080/api/v1/requests?
conversation=123e4567-e89b-12d3-a456-426614174000" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: user-james-bond" \
  -d '{
    "inputs": [
      {
        "role": "user",
        "content": [
          {
            "type": "goal",
            "value": "analyze_document",
            "payload": {
              "docType": "financial_report",
              "quarter": "Q3"
            }
          }
        ]
      }
    ]
  }'
```

---

4. Sending a Streaming Request

For real-time responses (typing effect), use the streaming endpoint.

- **Endpoint:** `POST /api/v1/requests/stream`
- **Response Content-Type:** `text/event-stream`

Example: Streaming Response

cURL

```
curl -X POST "http://localhost:8080/api/v1/requests/stream?
conversation=123e4567-e89b-12d3-a456-426614174000" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: user-james-bond" \
  --no-buffer \
  -d '{
    "inputs": [
      {
        "role": "user",
        "content": [{ "type": "text", "value": "Explain quantum
physics." }]
      }
    ]
  }'
```

---

5. Administrative Operations

To access Admin endpoints, the Gateway must inject the `admin` role in the `X-User-Roles` header.

Example: Fetching Global Statistics

cURL

```
curl -X GET "http://localhost:8080/api/v1/admin/stats/global" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: admin-user" \
  -H "X-User-Roles: admin"
```

Example: Listing LLM Provider Configurations

cURL

```
curl -X GET "http://localhost:8080/api/v1/admin/llm/provider-conf"
\
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: admin-user" \
  -H "X-User-Roles: admin"
```

Example: Creating an LLM Provider Configuration

cURL

```
curl -X POST "http://localhost:8080/api/v1/admin/llm/provider-conf"
\
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: admin-user" \
  -H "X-User-Roles: admin" \
  -d '{
    "provider": "openai",
    "defaultLlmModelConfName": "gpt5",
    "globalConf": {
      "apiSecret": "sk-your-api-key",
      "temperature": 0.7,
      "maxRetries": 3,
      "timeout": "60s"
    },
    "llModelConfs": [
      {
        "llmModelConfName": "gpt5",
        "modelName": "gpt-5.1",
        "multiModalSupported": true,
        "functionCallSupported": true
      },
      {
        "llmModelConfName": "gpt5-mini",
        "modelName": "gpt-5-mini",
        "temperature": 0.3,
        "multiModalSupported": true,
        "functionCallSupported": true
      }
    ]
  }'
```

For the complete list of LLM provider management endpoints, see REST API Reference — LLM Providers.

How-To Guides / How-To: Integrate AI Features in ARender

This guide details the process of adding context-aware AI buttons directly into the ARender High Content Interface (HCI). We will configure the necessary files to load the Uxopian-ai web components and add AI action buttons to the top panel.

Prerequisites

- **ARender HMI:** Installed and accessible.
- **Uxopian-ai:** Deployed and running.
- **Resources:** The `web-components.js` and `web-components.css` files are available.

File Structure Reference

The ARender configuration files should follow this structure:

```
├── configurations/
│   ├── arender-custom-client.properties  # Main configuration
entry point
│   ├── arender-plugins.xml               # Plugin loader
│   └── toppanel-arender-ai-configuration.xml # Button definitions
(Beans)
└── public/
    ├── web-components.css
    └── web-components.js
```

> **STARTER KIT**
>
> The Docker + ARender starter kit already includes these configuration files under `arender/configurations/`. You can use them as a starting point.

Step 1: Create the Prompts

The starter kit's `config/prompts.yml` already includes pre-defined prompts (e.g., `summarizeDocumentText`, `summarizeDocumentMarkdown`, `translate`, `detailedComparison`). If you need a custom prompt, create it via the API.

Endpoint: `POST /api/v1/admin/prompts`

Request Body (example):

```
{
  "id": "summarizeDocumentText",
  "role": "user",
  "content": "Summarize the following document in a plain text
format:\n\n[[${documentService.extractTextualContent(documentId)}]]",
  "defaultLlmProvider": "openai",
  "defaultLlmModel": "gpt-5.1",
  "temperature": "0.7",
  "timeSaved": 300,
  "requiresMultiModalModel": false,
  "requiresFunctionCallingModel": false
}
```

> **CONTEXT INJECTION**
>
> The expression `[[${documentService.extractTextualContent(documentId)}]]` indicates that the text extraction happens on the server side (Uxopian backend) using the `documentId` passed in the payload.

Step 2: Configure ARender Properties

Update `configurations/arender-custom-client.properties` to load the web components, define the menu button, and point to the AI host.

```
# 1. Load Styles
style.sheet=css/arender-style.css,web-components.css

# 2. Load the Web Component Script
arenderjs.startupScript=web-components.js

# 3. Add the 'aiMenu' to the top panel (middle section)
topPanel.section.middle.buttons.beanNames=addStickyNoteAnnotationButt

# 4. Configure the connection to Uxopian-ai
# Uses UXOPIAN_AI_HOST env var with a fallback default
uxopian.ai.host=${UXOPIAN_AI_HOST:http://localhost:8085}

# 5. Optional: Disable info toaster if preferred
toaster.log.info.enabled=false
```

> **⚠ UXOPIAN_AI_HOST**
>
> This must be the **public URL** reachable from the user's browser, not the internal Docker network address. Set it via the `UXOPIAN_AI_HOST` environment variable in your ARender UI container (see the Docker + ARender installation guide).

Step 3: Register the Plugin

Ensure ARender loads your custom bean configuration by importing it in `configurations/arender-plugins.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-lazy-init="true" default-autowire="no"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
beans.xsd">

    <import resource="plume.xml"/>
    <import resource="html-plugin.xml"/>
    <import resource="toppanel-arender-ai-configuration.xml"/>

</beans>
```

Step 4: Define the AI Buttons

Define the menu and buttons in `configurations/toppanel-arender-ai-configuration.xml`. The JavaScript handler invokes the `createChat` function exposed by the web component.

Below is a simplified example with a single "Summarize" button. The starter kit includes a more complete version with summarize, compare, translate, and open chat buttons.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-lazy-init="true" default-autowire="no"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd"

    <bean id="aiMenu"

class="com.arondor.viewer.client.toppanel.presenter.SubMenuButtonPres
       <constructor-arg value="aiMenu" />
       <constructor-arg value="AI" />
       <constructor-arg value="standardButton fas fa-robot toppanelBu
       <property name="enabled" value="true" />
       <property name="visibilityForTopPanel">
          <ref bean="topPanelVisibilityMode" />
       </property>
       <property name="orderedNamedList" value="summarizeTxtButton" /
    </bean>

    <bean id="summarizeTxtButton"

class="com.arondor.viewer.client.toppanel.presenter.DropdownMenuItemP
       <constructor-arg value="summarizeTxtButton"/>
       <constructor-arg value="Summarize in text format"/>
       <constructor-arg value="standardButton fas fa-list toppanelBut
       <property name="enabled" value="true" />
       <property name="closingOnClick" value="true" />
       <property name="buttonHandler">
          <bean
class="com.arondor.viewer.client.jsapi.toppanel.JSCallButtonHandler">
             <property name="jsCode">
                <value>
try {
    $wnd.createChat({
        endpoint: "${uxopian.ai.host}",
        wsEndpoint: "${uxopian.ai.host}",
```

```
        request: {
            inputs: [{
                role: 'user',
                content: [{
                    type: 'PROMPT',
                    value: 'summarizeDocumentText',
                    payload: {
                        documentId: $wnd.getARenderJS().getCurrentDoc
                    }
                }]
            }]
        }
    });
} catch(e) {
  console.log('Error : ' + e);
}
                </value>
              </property>
            </bean>
          </property>
    </bean>
</beans>
```

The `value` field in the JavaScript must match a prompt ID defined in the backend (here, `summarizeDocumentText` from `config/prompts.yml`).

Step 5: Build and Deploy

To deploy these changes, you must build a custom Docker image that extends the official ARender base image. This ensures your configuration files and web resources are correctly placed in the container's runtime environment.

STARTER KIT ALTERNATIVE

The starter kit's `uxopian-ai-stack.yml` already mounts `arender/configurations/` as a volume into the ARender UI container, so you can iterate without rebuilding the image during development.

Dockerfile Configuration

For production, create a `Dockerfile` at the root of your project:

```
# Stage 1: Preparation
FROM alpine as builder
WORKDIR /app
COPY configurations/ configurations/
COPY public/ public/

# Stage 2: Final Image
FROM artifactory.arondor.cloud:5001/arender-ui-springboot:2023.16.0

# Install configurations
COPY --from=builder /app/configurations/*
/home/arender/configurations/

# Install Web Resources
COPY --from=builder /app/public/* /home/arender/public/
```

Build Command

```
docker build -t my-custom-arender:1.0 .
```

How-To Guides / How-To: Integrate AI Features in FlowerDocs

This guide details the process of adding a context-aware AI feature—specifically a **Document Summarizer**—directly into the FlowerDocs user interface.

Prerequisites

Before proceeding, ensure the following environment configurations are in place:

- **Uxopian-ai Platform:** The core platform is deployed and running.
- **Web Component Resources:** The `uxopian-ai` web component JavaScript and CSS files are deployed and accessible to FlowerDocs.
- **BFF Configuration:** The Backend-for-Frontend (BFF) is configured to proxy requests correctly.
- **Plugin Setup:** The necessary plugin configurations (security, routing) have been established by the environment administrator.

Step 1: Create the Prompt

First, we must define the behavior of the AI by creating a specific prompt in the Uxopian-ai backend. This prompt will be called by the frontend script.

We will create a prompt with the ID `summarizeDocumentMarkdown`.

Endpoint: `POST /api/v1/admin/prompts`

Request Body:

```
{
  "id": "summarizeDocMd",
  "role": "user",
  "content": "Summarize the following document in a plain text
format: \n
[[${documentService.extractTextualContent(documentId)}]]",
  "defaultLlmProvider": "openai",
  "defaultLlmModel": "gpt-5.1",
  "temperature": "0.7",
  "timeSaved": 300,
  "requiresMultiModalModel": false,
  "requiresFunctionCallingModel": false
}
```

> **VARIABLES**
>
> Notice the usage of `[[${documentId}]]` in the content. This variable is resolved server-side by the Thymeleaf engine using the `documentId` value passed in the request payload (see Step 2).

Step 2: Configure the FlowerDocs Script

Next, navigate to the **FlowerDocs Administration Console** and access the **Scripts** section. Create a new JavaScript file (e.g., `Summarize Document`) and

paste the following code.

This script uses an **IIFE (Immediately Invoked Function Expression)** to encapsulate logic and prevent global variable conflicts.

```
(function () {
  /**
   * ================================================================
   * 1. CONFIGURATION & CONSTANTS
   * Constants are scoped to this function to prevent global
conflicts.
   * ================================================================
   */
  const BASE_URL = window.location.origin;
  const ENDPOINTS = {
    CHAT: `${BASE_URL}/gui/gateway/uxopian-ai`,
    WS: `${BASE_URL}/gui/gateway/uxopian-ai`,
    // Dynamically retrieve the gateway URL based on the current
user scope
    getGATEWAY: () =>
      `${BASE_URL}/gui/plugins/${JSAPI.get()
        .getUserAPI()
        .getScope()}/gateway/uxopian-ai`,
  };

  /**
   * ================================================================
   * 2. HELPER FUNCTIONS
   * ================================================================
   */

  /**
   * Generates the technical context (JSON) required by the AI.
   * This injects the current FlowerDocs Component ID and ARender
Document ID.
   */
  function getComponentContext() {
    const flowerId = JSAPI.get()
      .getLastComponentFormAPI()
      .getComponent()
      .getId();
    const arenderId = arenderJSAPI.getCurrentDocumentId();
```

```
    return InputBuilder.textAsSystem(`
      /* Contextual Data */
      {
        flowerdocId: {
          value: "${flowerId}",
          description: "The ID of the Flowerdoc document"
        },
        arenderDocId: {
          value: "${arenderId}",
          description: "The ID of the Arender document (BASE64
encoded)"
        }
      }
    `);
  }

  /**
   * Initiates the AI Chat interface using the Web Component's
exposed function.
   */
  function openChatWindow(requestPayload) {
    fetch(ENDPOINTS.getGATEWAY())
      .then(() =>
        createChat({
          endpoint: ENDPOINTS.CHAT,
          wsEndpoint: ENDPOINTS.WS,
          request: requestPayload,
        })
      )
      .catch((error) => console.error("Failed to open chat:",
error));
  }

  /**
   * Registers a custom button in the application header.
   */
  function registerHeaderAction({ label, icon, onExecute }) {
```

```
    const jsapi = JSAPI.get();

    // Register the label for translation
    jsapi.getLabelsAPI().setLabels([label]);

    jsapi.registerForComponentChange((api) => {
      const resolvedLabel =
jsapi.getLabelsAPI().getLabel(label.name);
      const headerActions = api.getActions().getHeaderActions();

      const actionItem = jsapi
        .getActionFactoryAPI()
        .buildMenu(
          `ai-action-${label.name}`,
          resolvedLabel,
          icon || "fa-solid fa-robot",
          onExecute
        );

      headerActions.add(actionItem);
    });
  }

  /**
   * ================================================================
   * 3. MAIN EXECUTION
   * ================================================================
   */
  registerHeaderAction({
    icon: "fa fa-file-alt",
    label: {
      name: "summarizeDocMd",
      FR: "Résumer le document en Markdown",
      EN: "Summarize the document in Markdown",
    },
    // The onExecute function builds the request at the moment of
the click
    onExecute: () => {
```

```
      const request = {
        inputs: [
          // 1. Inject System Context (IDs)
          getComponentContext(),
          // 2. Inject User Prompt with dynamic Arender ID
          InputBuilder.promptAsUser("summarizeDocMd", {
            documentId: arenderJSAPI.getCurrentDocumentId(),
          }),
        ],
      };
      openChatWindow(request);
    },
  });
})();
```

Code Breakdown

Context Injection (`getComponentContext`)

The script automatically retrieves the current context from the FlowerDocs UI (`JSAPI`) and the viewer (`arenderJSAPI`). It formats this data as a **System Message**, ensuring the LLM understands which specific document it is analyzing.

Dynamic Execution (`onExecute`)

The request body is constructed inside the `onExecute` callback. This ensures that the `documentId` and context are fetched **at the moment the button is clicked**, rather than when the page loads. This is crucial for Single Page Applications (SPAs) where the user might switch documents without reloading the page.

The Chat Trigger (`createChat`)

The `createChat` function is exposed globally by the `uxopian-ai` web component. It accepts the connection endpoints and the initial `request` payload, seamlessly opening the chat modal over the FlowerDocs interface.

How-To Guides / How-To: Integrate AI into a Basic Web Page

This guide explains how to add Uxopian-ai features to a standard HTML/JavaScript application. We will create a simple button that, when clicked, opens the AI chat modal with specific context from the page.

Prerequisites

- **Uxopian-ai Platform:** Deployed and accessible.
- **Web Components:** The `web-components.js` and `web-components.css` files must be imported into your HTML file.

Step 1: Create the Prompt

First, configure the prompt in the backend. This prompt will define how the AI behaves and what variables it expects from the frontend (e.g., the page title or user name).

Endpoint: `POST /api/v1/admin/prompts`

Request Body:

```
{
  "id": "webAssistantPrompt",
  "role": "system",
  "content": "You are a helpful assistant for the website. The user
is currently viewing the page: '[[${pageTitle}]]'. Answer their
questions concisely.",
  "defaultLlmProvider": "openai",
  "defaultLlmModel": "gpt-5.1",
  "temperature": "0.7",
  "timeSaved": 60,
  "requiresMultiModalModel": false,
  "requiresFunctionCallingModel": false
}
```

Step 2: HTML Structure

Create a basic HTML file. You must link the Uxopian CSS and JS files. Add a button that will trigger the interaction.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>My Web App</title>

    <link rel="stylesheet" href="web-components.css" />
  </head>
  <body>
    <header>
      <h1>Welcome to My App</h1>
      <button id="ask-ai-btn">Ask AI Helper</button>
    </header>

    <main>
      <p>This is a sample page content.</p>
    </main>

    <script src="web-components.js"></script>

    <script src="app.js"></script>
  </body>
</html>
```

Step 3: JavaScript Implementation

In your `app.js` file, add an event listener to the button. When clicked, it will call the `createChat` function exposed by the web component.

We will verify that the prompt ID matches the one created in Step 1, and we will pass the document title dynamically via the `payload`.

```
// Configuration
const AI_CONFIG = {
  ENDPOINT: "https://your-uxopian-instance.com", // HTTP Endpoint
  WS_ENDPOINT: "https://your-uxopian-instance.com", // WebSocket
Endpoint (usually same base)
};

document.addEventListener("DOMContentLoaded", () => {
  const aiButton = document.getElementById("ask-ai-btn");

  if (aiButton) {
    aiButton.addEventListener("click", () => {
      // 1. Gather Context (e.g., page title, selected text, user
ID)
      const currentPageTitle = document.title;

      // 2. Trigger the Chat Interface
      try {
        // 'createChat' is globally available via web-components.js
        createChat({
          endpoint: AI_CONFIG.ENDPOINT,
          wsEndpoint: AI_CONFIG.WS_ENDPOINT,
          request: {
            inputs: [
              {
                role: "user",
                content: [
                  {
                    type: "PROMPT",
                    value: "webAssistantPrompt", // Must match
Backend ID
                    payload: {
                      // Dynamic variables to inject into the
prompt
                      pageTitle: currentPageTitle,
                    },
                  },
```

```
                ],
              },
            ],
          },
        });
      } catch (error) {
        console.error(
          "Uxopian-ai Web Component not loaded or error
initializing:",
          error
        );
      }
    });
  }
});
```

Technical Details

The `createChat` Function

This function allows you to open the chat modal programmatically.

- **endpoint:** The base URL for the REST API (e.g., for retrieving history).
- **wsEndpoint:** The base URL for WebSocket connections (used for streaming responses).
- **request:** The initial payload sent to the LLM to start the conversation.
  - **inputs:** An array of message objects.
  - **value:** The ID of the prompt configuration on the server.
  - **payload:** A key-value object. Keys must match the variables defined in your prompt (e.g., `[[${pageTitle}]]`).

CORS CONFIGURATION

Ensure your Uxopian-ai backend is configured to allow Cross-Origin Resource Sharing (CORS) from your web application's domain.

How-To Guides / 🛡️ Backup and Recovery

This section provides guidance on how to protect the critical data managed by the `uxopian-ai` framework. A proper backup strategy is essential to prevent data loss and ensure business continuity.

💾 What to Back Up

There are two primary sources of data you need to protect:

- 🛠️ **Configuration Files:** All the `.yml` files that define the service's behavior, including connections, provider settings, and default parameters.
- 📦 **OpenSearch Data:** The OpenSearch instance stores all the dynamic data generated by user interactions, which includes conversations, messages, and the centrally managed Prompts and Goals.

🧠 Built-in YAML Backup for Prompts and Goals

The framework includes an automatic backup mechanism for Prompts and Goals. Whenever you create, update, or delete a prompt or goal via the API, the service automatically writes the current state to YAML files in the configured backup directory.

> **Note:** This mechanism is multi-tenant aware, meaning it generates separate files for each tenant to ensure data isolation.

⚙️ Configuration

You only need to configure the directory path where backups should be stored.

For Prompts (`prompts.yml`):

```
prompts:
  backup:
    path: ${PROMPTS_BACKUP_PATH:./prompts/}
```

For Goals (`goals.yml`):

```
goals:
  backup:
    path: ${GOALS_BACKUP_PATH:./goals/}
```

📂 File Naming Convention

The system automatically names backup files based on the `tenantId`.

Format:

- **Prompts:** `prompts-{tenantId}.yaml`
- **Goals:** `goals-{tenantId}.yaml`

Content Structure: A backup file contains the raw definition of a tenant's configuration.

```
tenantId: Tenant-development
prompts:
  - id: basePrompt
    role: SYSTEM
    content: "..."
    # ...
```

🔄 Restoring from YAML Backups

Since backup files contain specific tenant data, restoration involves integrating this data back into your main configuration files or pushing it via the API.

✅ Method 1: Restoration via Configuration (Bootstrap)

This method allows you to restore data by injecting the backup content directly into the main configuration files (`prompts.yml` or `goals.yml`) before starting the service.

Steps:

1. **Locate Backup:** Open the specific backup file you wish to restore (e.g., `prompts-Tenant-development.yaml`).
2. **Copy Content:** Copy the entire content of the file (including `tenantId` and the list of prompts/goals).
3. **Edit Main Config:** Open your main configuration file (`config/prompts.yml` or `config/goals.yml`).
4. **Inject and Configure:**
   - Locate the `tenants: []` list.

- Paste the backup content as a new item in this list.
- **Crucial:** Add the `mergeStrategy` field to define how this data should be applied.

Example for Prompts (`config/prompts.yml`):

```
prompts:
  globals: [...]

  # Paste your backup here under 'tenants'
  tenants:
    - tenantId: Tenant-development # <--- From Backup
      mergeStrategy: OVERRIDE # <--- ADD THIS MANUALLY
      prompts: # <--- From Backup
        - id: basePrompt
          role: SYSTEM
          content: "..."
```

5. **Restart:** Restart `uxopian-ai`. The service will process the tenants list and apply the configuration to OpenSearch according to the `mergeStrategy`.

🧪 Method 2: Manual Restore via API

You can also manually restore a specific tenant's configuration using the REST API.

1. Open the backup file (e.g., `goals-Tenant-development.yaml`).
   - **Goals:** Copy each entry from the `goalGroups` list.
   - **Prompts:** Copy each entry from the `prompts` list.
2. Send a `POST` request for each item to the appropriate endpoint.
   - **Prompts:** `POST /api/v1/admin/prompts`
   - **Goals:** `POST /api/v1/admin/goals`

3. **Headers:** You must provide the correct authentication headers to ensure the data is restored to the correct tenant.
    - `X-User-TenantId: Tenant-development`

🧩 Merge Strategies (Configuration Mode)

When using **Method 1**, the `mergeStrategy` field controls how the restored data interacts with existing data in OpenSearch.

🔁 OVERRIDE (Disaster Recovery)

- **Behavior:** Deletes **ALL** existing prompts/goals for this specific tenant in OpenSearch and replaces them with the content of the YAML.
- **Use Case:** Restoring a corrupted environment or reverting to a known clean state.

➕ MERGE (Update/Patch)

- **Behavior:**
    - Updates items with matching IDs.
    - Creates items that do not exist.
    - Does **not** delete items that are in OpenSearch but not in the YAML.
- **Use Case:** Applying configuration updates without losing data created by users since the backup.

🆕 CREATE_IF_MISSING (Safe Init)

- **Behavior:** Only creates items that do not currently exist. Existing items are ignored.
- **Use Case:** Deploying default prompts to a new environment without risking overwrite of existing data.

🗂️ Backup Strategy Summary

Entity	Storage	Backup Mechanism	Recommended Restore Method
Configuration	Filesystem	Copy files manually	Restore files to `./config/`
Conversations	OpenSearch	OpenSearch Snapshots	Elastic/OpenSearch Snapshot Restore
Prompts/Goals	OpenSearch	Auto YAML Backup	Method 1 (Config Merge) for full restore

---

✅ Key Recommendations

- 📅 **Automate Commits:** Since the system automatically updates the YAML files on disk, set up a cron job to commit and push these changes to a Git repository automatically.
- 📝 **Verify Merge Strategy:** When restoring via config, always double-check the `mergeStrategy`. Using `OVERRIDE` by mistake will wipe out any new prompts created since the backup.

How-To Guides / Monitoring Performance

This guide covers how to monitor the health and performance of your uxopian-ai deployment using the built-in observability stack.

Monitoring Architecture

The monitoring system operates in two distinct stages:

1. **Data Collection (Actuator)**: The Spring Boot Actuator module exposes operational information about the running application — health, loggers, info.
2. **Instrumentation (Micrometer)**: Micrometer, an application metrics facade, instruments key methods with `Timer` metrics, recording the duration and count of every execution for a clear view of API performance.

Actuator Endpoints

Spring Boot Actuator exposes the following endpoints (configured in `metrics.yml`):

Endpoint	Description
`/actuator/health`	Application health status.
`/actuator/info`	General application information.
`/actuator/loggers`	View and modify log levels at runtime.

Checking Health

```
curl http://localhost:8080/actuator/health
```

A healthy response returns:

```
{
  "status": "UP"
}
```

Metrics Export to OpenSearch

Uxopian-ai uses Micrometer to export telemetry data directly to your OpenSearch instance. This data powers the Admin Dashboard statistics.

Configuration

The metrics export is configured in `metrics.yml`:

```
management:
  elastic:
    metrics:
      export:
        enabled: true
        host: http://${opensearch.host}:${opensearch.port}
        index: micrometer-metrics
        auto-create-index: true
  endpoints:
    web:
      exposure:
        include: health,info,loggers
  metrics:
    uxopian-ai:
      enable: true  # Enables custom business metrics (Token usage,
ROI, etc.)
```

Custom Business Metrics

When `management.metrics.uxopian-ai.enable` is set to `true`, the following business metrics are collected:

- **Token consumption** — Input and output tokens per request.
- **Request latency** — Duration of LLM calls.
- **ROI tracking** — Time saved per prompt usage.
- **Feature adoption** — Multi-modal and function calling usage rates.

These metrics feed directly into the Statistics dashboard.

Disabling Noisy Metrics

By default, standard JVM and framework metrics can generate a large volume of data. You can disable them selectively in `metrics.yml`:

```
management:
  metrics:
    enable:
      application: false
      tomcat: false
      logback: false
      jvm: false
      system: false
      http: false
      process: false
      disk: false
      executor: false
```

Recommended Monitoring Setup

For production deployments, we recommend:

1. **Health checks**: Point your load balancer or orchestrator at `/actuator/health`.
2. **Log aggregation**: Forward application logs to a centralized system (ELK, Datadog, etc.).
3. **Metrics visualization**: Use the built-in Admin Dashboard or connect an external tool (Grafana, Kibana) to the `micrometer-metrics` index in OpenSearch.
4. **Alerting**: Set up alerts on key metrics — LLM response latency, error rates, and token consumption spikes.

Extending Uxopian-ai / How to Create Custom Prompt Helpers

Helpers are custom Java services that allow you to inject dynamic data or perform server-side operations directly within your LLM prompts.

Overview

Uxopian-ai uses the **Thymeleaf** engine to interpret prompts. By creating a custom Helper, you expose Java objects and methods that can be called using the standard Thymeleaf syntax: `[[${helperName.methodName()}]]`.

Common Use Cases:

- **Data Enrichment:** Fetching real-time data (Weather, Stock prices, Time).
- **Content Retrieval:** Calling external systems (e.g., FlowerDocs) to retrieve document text or images.
- **Data Transformation:** Formatting dates, hashing strings, or parsing complex inputs before sending them to the LLM.

Prerequisites

Before starting, ensure your development environment is configured to access the Uxopian artifacts.

1. **Maven Settings:** You must configure your `.m2/settings.xml` to include the credentials and repository definitions required to download the `com.uxopian.ai` libraries.
2. **Java Development Kit (JDK):** Ensure you are using a compatible JDK version (Java 21+ recommended).

Step 1: Project Configuration

Create a new Maven project or module. You need to import the `shared` parent project and the `annotation` dependency to mark your services correctly.

Add the following to your `pom.xml` (adjust the `<version>` to match your current Uxopian-ai deployment):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="[http://maven.apache.org/POM/4.0.0]
(http://maven.apache.org/POM/4.0.0)"
         xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance]
(http://www.w3.org/2001/XMLSchema-instance)"
         xsi:schemaLocation="[http://maven.apache.org/POM/4.0.0]
(http://maven.apache.org/POM/4.0.0)
[http://maven.apache.org/xsd/maven-4.0.0.xsd]
(http://maven.apache.org/xsd/maven-4.0.0.xsd)">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>my-custom-helper</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.uxopian.ai</groupId>
            <artifactId>annotation</artifactId>
        </dependency>
        </dependencies>
</project>
```

Step 2: Implement the Service

To create a helper, you must create a Java class annotated with `@Component` (standard Spring) and `@HelperService`.

The `name` attribute defined in `@HelperService` is the **variable name** you will use in your prompt.

```
package com.mycompany.uxopian.helpers;

import org.springframework.stereotype.Component;
import com.uxopian.ai.model.annotation.helper.HelperService;

@Component
@HelperService(name = "myTools") // This service will be accessible
as 'myTools' in prompts
public class MyCustomHelper {

    /**
     * Example method to get current status.
     * Usage in prompt: [[${myTools.getStatus()}]]
     */
    public String getStatus() {
        return "System is operational";
    }

    /**
     * Example method with parameters.
     * Usage in prompt: [[${myTools.greet(userName)}]]
     */
    public String greet(String name) {
        return "Hello, " + name + "!";
    }
}
```

Step 3: Building the Artifact (Fat JAR)

Because your helper might rely on third-party libraries that are not present in the core Uxopian-ai classpath, **you must package your helper as a Fat JAR (or Uber JAR)**.

This ensures all your specific dependencies are included inside the JAR file. You can achieve this using the `maven-shade-plugin`.

> **DEPENDENCY CONFLICTS**
>
> Ensure you do not bundle core Spring or Uxopian dependencies that conflict with the running server. Scope strictly necessary dependencies.

Step 4: Deployment

Uxopian-ai loads helpers from a specific directory on the filesystem, typically `helper-services/`.

Docker Deployment (Recommended)

When running Uxopian-ai in a containerized environment, the best practice is to build a custom Docker image that extends the base image and adds your JAR.

Dockerfile Example:

```
# Start from the specific version of Uxopian-ai
FROM artifactory.arondor.cloud:5001/uxopian-ai:v2026.0.0-ft1-rc2-full

# Copy your custom helper Fat JAR into the loader directory
COPY ./target/my-custom-helper-1.0-SNAPSHOT.jar /app/helper-services/
```

Step 5: Usage in Prompts & Requests

Once deployed, your helper is ready to be used in Prompts.

1. The Prompt Template

In your Thymeleaf prompt, call the method using the name defined in the annotation.

Example Prompt:

```
The system status is: [[${myTools.getStatus()}]]
Please write a polite message for:
[[${myTools.greet(customerName)}]]
```

2. Passing Parameters (The Payload)

If your helper method requires parameters (like `greet(String name)` above), the values for these parameters must be provided in the request payload.

Crucial Rule: The variable names in the payload **must match** the variable names used in the method call within the prompt.

API Request Example: If your prompt contains `[[${myTools.greet(customerName)}]]`, your JSON payload to the request endpoint must include `customerName`:

```
{
  "conversation": "...",
  "inputs": [
    {
      "role": "user",
      "content": "...",
      "payload": {
        "customerName": "Alice"
      }
    }
  ]
}
```

TROUBLESHOOTING

If your helper returns null or throws an error, verify that: 1. The payload key matches the argument name used in the Thymeleaf expression exactly. 2. The variable type in the payload matches the method signature (e.g., Integer vs String).

Extending Uxopian-ai / Advanced Helpers: Map-Reduce Paradigm

This guide explains how to implement **Advanced Helpers** in Uxopian-ai using a **Map-Reduce paradigm** to summarize or process large documents exceeding standard LLM context limits.

Overview

The Map-Reduce approach allows splitting heavy tasks into manageable chunks processed in parallel (Map) and then combining and reducing the results recursively (Reduce).

Workflow:

1. **Client Request**: Triggers the main user prompt.
2. **Main Prompt**: Invokes the Advanced Helper.
3. **Helper (Map Phase)**: Splits document into chunks and processes them in parallel using an internal LLM prompt.
4. **Helper (Reduce Phase)**: Aggregates results and recursively condenses until final summary fits a single context window.
5. **Output**: Returns final summarized or processed content to the Main Prompt.

Step 1: Define Internal Map Prompt

Register an internal prompt that the Helper will call for each chunk of data.

```
curl -X POST "http://localhost:8080/api/v1/admin/prompts" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Roles: admin" \
  -d '{
    "id": "map_chunk_prompt",
    "role": "system",
    "content": "Role: Expert Analyst.\nGoal: Extract key facts
efficiently.\nInstructions:\n1. Extract critical points as bullet
list.\n2. Merge duplicates.\n3. Output only the content.\n\nInput
Data:\n[[${content}]]",
    "defaultLlmProvider": "openai",
    "defaultLlmModel": "gpt-5.1",
    "timeSaved": 5
  }'
```

> **Recommendation:** Use a fast, cheap model for the map phase (e.g., `gpt-5-mini` or `gpt-4.1-nano`). These intermediate steps process many chunks in parallel — using a flagship model here would multiply costs with little quality benefit. Reserve the powerful model for the final reduce/synthesis step (Step 3). See Choosing the Right Model for the full model tier guide.

Step 2: Implement the Advanced Helper (Java)

The Helper orchestrates splitting, parallel processing, and recursive reduction.

```
@Service
@HelperService(name = "advancedMapReduceHelper")
public class AdvancedMapReduceHelper {

    private static final String INTERNAL_PROMPT_ID =
"map_chunk_prompt";
    private final ExecutorService executorService =
Executors.newVirtualThreadPerTaskExecutor();
    private final SecureRequestService requestService;

    public String processDocument(String docId) {
        List<String> rawContents = fetchDocumentContent(docId);
        ContextSnapshot contextSnapshot =
AiContext.captureSnapshot();
        try (AiResourceContext ignored = contextSnapshot.restore())
{
            return recursiveProcess(rawContents, contextSnapshot);
        }
    }

    private String recursiveProcess(List<String> segments,
ContextSnapshot contextSnapshot) {
        if (fitsInOneContext(segments)) {
            return callLlm(String.join("\n", segments),
contextSnapshot);
        }

        List<String> chunks = createOptimizedChunks(segments);
        List<CompletableFuture<String>> futures = chunks.stream()
            .map(chunk -> CompletableFuture.supplyAsync(() -> {
                try (AiResourceContext ignored =
contextSnapshot.restore()) {
                    return callLlm(chunk, contextSnapshot);
                }
            }, executorService))
            .toList();
```

```
        List<String> results =
futures.stream().map(CompletableFuture::join).toList();
        return recursiveProcess(results, contextSnapshot);
    }

    private String callLlm(String text, ContextSnapshot
contextSnapshot) {
        LlmConfig conf = new LlmConfig();
        conf.setDisableReasoning(true);
        return
requestService.sendRequestsWithoutHistory(buildRequest(text), conf,
currentUser).getAnswer();
    }
}
```

Step 3: Create Main User Prompt

The user-facing prompt triggers the Advanced Helper and specifies output constraints.

```
curl -X POST "http://localhost:8080/api/v1/admin/prompts" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Roles: admin" \
  -d '{
    "id": "process_long_document",
    "role": "user",
    "content": "Please summarize the document in 500
words:\n\n[[${advancedMapReduceHelper.processDocument(documentId)}]]"
    "defaultLlmProvider": "openai",
    "defaultLlmModel": "gpt-5.1",
    "timeSaved": 60
  }'
```

> **Helper Call:**
> `[[${advancedMapReduceHelper.processDocument(documentId)}]]`

Step 4: Usage via API

Invoke the prompt and provide the document ID.

```
curl -X POST "http://localhost:8080/api/v1/requests?
conversation=uuid" \
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Id: user-id" \
  -d '{
    "inputs": [
      {
        "role": "user",
        "content": [
          {
            "type": "prompt",
            "value": "process_long_document",
            "payload": {"documentId": "doc-12345"}
          }
        ]
      }
    ]
  }'
```

> **RESULT**

1. System loads `process_long_document` template.

2. Advanced Helper executes Map-Reduce over document chunks.
3. Intermediate results are recursively merged.
4. Final summary is returned as a concise 500-word narrative.

Extending Uxopian-ai / How to Create Custom Prompt Tools

Tools

Tools are custom Java services that empower the LLM to perform actions or retrieve information autonomously. Unlike **Helpers** (which inject data into the prompt before generation), **Tools** are *Function Calling* capabilities that the AI can decide to execute during the conversation.

Overview

Uxopian-ai uses **LangChain4j** to manage Tools. By creating a custom Tool, you provide the LLM with a set of methods (functions) described in natural language. The LLM analyzes the user's request and determines if and when to call your Java methods.

Common Use Cases

- **Action Execution**: Sending emails, creating Jira tickets, updating a database.
- **Dynamic Queries**: Searching an SQL database based on natural language criteria.

- **Complex Calculations**: Performing mathematical operations that LLMs struggle with.

Prerequisites

Before starting, ensure your development environment is configured properly.

- **Maven Settings**: Configure your `.m2/settings.xml` to access the `com.uxopian.ai` libraries.
- **Java Development Kit (JDK)**: Ensure you are using a compatible JDK version (**Java 21+ recommended**).

Step 1: Project Configuration

Create a new Maven project. You need to import the **Uxopian annotation** dependency (to mark the service) and the **langchain4j-core** dependency (to define the tool methods).

Add the following to your `pom.xml`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>my-custom-tool</artifactId>

    <properties>
        <langchain4j.version>1.11.0</langchain4j.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.uxopian.ai</groupId>
            <artifactId>annotation</artifactId>
        </dependency>

        <dependency>
            <groupId>dev.langchain4j</groupId>
            <artifactId>langchain4j-core</artifactId>
            <version>${langchain4j.version}</version>
        </dependency>
    </dependencies>
</project>
```

Step 2: Implement the Service

The **Uxopian-ai Tool Loader** scans for classes annotated with `@ToolService`. These classes must also be standard Spring beans (annotated with `@Component` or `@Service`).

1. The Tool Class

You must use the `@Tool` annotation from LangChain4j on the methods you want to expose to the AI. The text inside `@Tool("...")` is crucial: it is the description the LLM reads to understand what the tool does.

```
package com.mycompany.uxopian.tools;

import org.springframework.stereotype.Service;
import com.uxopian.ai.model.annotation.tool.ToolService;
import dev.langchain4j.agent.tool.Tool;

@Service("bookingTool") // Standard Spring annotation
@ToolService // Marks this as a Tool for Uxopian-ai scanning
public class BookingTool {

    private static final org.slf4j.Logger logger =
org.slf4j.LoggerFactory.getLogger(BookingTool.class);

    /**
     * The description inside @Tool is what the LLM sees.
     * Be descriptive about what the method does and what the
parameters represent.
     */
    @Tool("Checks the availability of a meeting room for a specific
date")
    public boolean checkAvailability(String roomName, String date)
{
        // Logic to check database or external API
        logger.info("Checking availability for {} on {}", roomName,
date);
        return true;
    }

    @Tool("Books a meeting room if available")
    public String bookRoom(String roomName, String date, String
organizer) {
        return "Room " + roomName + " successfully booked for " +
organizer + " on " + date;
    }
}
```

2. Internal Dependencies (Optional)

The `ToolServiceLoader` also scans and registers internal beans found in your JAR. If your Tool relies on a repository or a helper class, simply annotate them with `@Component` or `@Service` (without `@ToolService`), and they will be injected automatically.

```
@Component
public class InternalDatabaseConnector {
    // This bean is not exposed to the LLM, but can be Autowired
into BookingTool
}
```

Step 3: Building the Artifact (Fat JAR)

Just like Helpers, Tools must be packaged as a **Fat JAR (Uber JAR)** to include all specific dependencies (e.g., database drivers, specific HTTP clients) that are not part of the core platform.

When building your Fat JAR, you **must exclude** `langchain4j-core` (and other platform-provided libraries) to avoid classpath conflicts at runtime.

Maven Shade Plugin Example

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.5.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <artifactSet>
                            <excludes>

<exclude>dev.langchain4j:langchain4j-core</exclude>
                            </excludes>
                        </artifactSet>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

> ⚠️ **Dependency Conflicts** Be careful not to include `langchain4j-core` or Spring framework JARs in your final artifact if they are already provided by the platform, unless you specifically need to override them (which is **not recommended**).

Step 4: Deployment

Uxopian-ai scans a specific directory for tools, typically configured as `tools/`.

Docker Deployment (Recommended)

Add your compiled JAR to the `tools/` directory in your custom Docker image.

Dockerfile example:

```
# Start from the specific version of Uxopian-ai
FROM artifactory.arondor.cloud:5001/uxopian-ai:v2026.0.0-ft1-rc2-full

# Copy your custom Tool Fat JAR into the tools directory
COPY ./target/my-custom-tool-1.0-SNAPSHOT.jar /app/tools/
```

Step 5: Usage

Unlike Helpers, you do **not** call Tools explicitly in your prompt (e.g., no `[[${...}]]`).

1. **Enable the Tool**: In the Uxopian-ai configuration (or Assistant setup), ensure your new Tool is selected/enabled for the conversation context.
2. **Prompting**: Simply ask the LLM to perform the task.

Example Scenario

User says:

> "Can you book the 'Red Room' for me for tomorrow?"

LLM process:

1. The LLM reads the `@Tool` description: "*Books a meeting room if available*".
2. It extracts the parameters:
   - `roomName = "Red Room"`
   - `date = "2025-10-20"`
   - `organizer = "User"`
3. It executes the Java method `bookRoom(...)` server-side.
4. It receives the returned `String`.

LLM response:

> "I have successfully booked the Red Room for you for tomorrow."

Best Practices for Descriptions

The quality of your Tool depends directly on the quality of your `@Tool` annotation descriptions.

Bad:

```
@Tool("Get data")
```

Good:

```
@Tool("Retrieves the current stock price for a given ticker symbol
(e.g., AAPL)")
```

Extending Uxopian-ai / Adding a New LLM Provider (Custom Connector)

Uxopian-ai is built on an extensible architecture that allows you to integrate any Large Language Model (LLM).

While the system uses **LangChain4j** as its internal abstraction layer, you are not limited to existing adapters. You can build a **fully custom connector** to integrate:

- Internal/Local models hosted on your infrastructure.
- Proprietary APIs not supported by standard libraries.
- Mock/Test services for development.

This guide demonstrates how to implement a connector from scratch using a hypothetical **"FakeLLM"** that simulates responses.

Prerequisites

- **Java 21+**: The LLM Connector architecture requires Java 21 or higher.
- **Maven Settings**: Your `.m2/settings.xml` must be configured to access Uxopian-ai artifacts.

Step 1: Project Configuration

Create a new Maven module. Inherit from the `shared` parent and include the `llm-connector` artifact.

`pom.xml` Setup:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="[http://maven.apache.org/POM/4.0.0]
(http://maven.apache.org/POM/4.0.0)"
         xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance]
(http://www.w3.org/2001/XMLSchema-instance)"
         xsi:schemaLocation="[http://maven.apache.org/POM/4.0.0]
(http://maven.apache.org/POM/4.0.0)
[http://maven.apache.org/xsd/maven-4.0.0.xsd]
(http://maven.apache.org/xsd/maven-4.0.0.xsd)">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>my-custom-connector</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.uxopian.ai</groupId>
            <artifactId>llm-connector</artifactId>
        </dependency>
        </dependencies>
</project>
```

Step 2: Implement the Chat Logic (`ChatModel`)

To create a custom connector, you must implement the `dev.langchain4j.model.chat.ChatModel` interface. This is where the core logic resides: receiving a request, processing it (or calling an external API), and returning a standardized response.

Example: `FakeChatModel.java`

```
package com.uxopian.ai.fakellm.client;

import java.util.List;
import dev.langchain4j.data.message.AiMessage;
import dev.langchain4j.data.message.ChatMessage;
import dev.langchain4j.data.message.UserMessage;
import dev.langchain4j.model.chat.ChatModel;
import dev.langchain4j.model.chat.request.ChatRequest;
import dev.langchain4j.model.chat.response.ChatResponse;
import dev.langchain4j.model.output.FinishReason;
import dev.langchain4j.model.output.TokenUsage;

public class FakeChatModel implements ChatModel {

    private final String modelName;

    public FakeChatModel(String modelName) {
        this.modelName = modelName;
    }

    @Override
    public ChatResponse chat(ChatRequest request) {
        // 1. Extract the last message from the user
        String userText = findLastUserMessage(request.messages());

        // 2. Simulate Generating a Response (In a real scenario,
call your API here)
        String fakeResponse = "FakeLLM (" + modelName + ") says: I
received your message: '" + userText + "'";

        // 3. Calculate simulated token usage
        TokenUsage usage = new TokenUsage(10, 20, 30);

        // 4. Return the standardized ChatResponse
        return ChatResponse.builder()
                .modelName(modelName)
                .tokenUsage(usage)
```

```
                .aiMessage(AiMessage.from(fakeResponse))
                .finishReason(FinishReason.STOP)
                .build();
    }

    // Helper to find the text input
    private String findLastUserMessage(List<ChatMessage> messages)
{
        return messages.stream()
                .filter(UserMessage.class::isInstance)
                .map(m -> ((UserMessage) m).singleText())
                .reduce((first, second) -> second)
                .orElse("Hello");
    }
}
```

Step 3: Implement Streaming (`StreamingChatModel`)

You must also implement `StreamingChatModel`. This interface allows the backend to stream tokens to the user interface in real-time.

Example: `FakeStreamingChatModel.java`

```
package com.uxopian.ai.fakellm.client;

import dev.langchain4j.data.message.AiMessage;
import dev.langchain4j.model.chat.StreamingChatModel;
import dev.langchain4j.model.chat.request.ChatRequest;
import dev.langchain4j.model.chat.response.ChatResponse;
import
dev.langchain4j.model.chat.response.StreamingChatResponseHandler;
import dev.langchain4j.model.output.FinishReason;
import dev.langchain4j.model.output.TokenUsage;

public class FakeStreamingChatModel implements StreamingChatModel {

    private final String modelName;

    public FakeStreamingChatModel(String modelName) {
        this.modelName = modelName;
    }

    @Override
    public void chat(ChatRequest request,
StreamingChatResponseHandler handler) {
        try {
            // Simulate a streaming response by sending chunks with
a delay
            String[] chunks = {"Hello ", "this ", "is ", "Fake ",
"Streaming..."};
            StringBuilder fullResponse = new StringBuilder();

            for (String chunk : chunks) {
                // Emit partial token
                handler.onPartialResponse(chunk);
                fullResponse.append(chunk);

                // Simulate network latency
                Thread.sleep(200);
            }
```

```
            // Finalize the stream
            TokenUsage usage = new TokenUsage(10, 10, 20);
            ChatResponse response = ChatResponse.builder()

.aiMessage(AiMessage.from(fullResponse.toString()))
                    .tokenUsage(usage)
                    .finishReason(FinishReason.STOP)
                    .build();

            handler.onCompleteResponse(response);

        } catch (Exception e) {
            handler.onError(e);
        }
    }
}
```

Step 4: Create the Provider Service

Finally, create the **Service** that registers your provider with Uxopian-ai. This class implements `ModelProvider` and acts as a factory for your models.

Crucial: You must annotate this class with `@Service("your-provider-name")`. This name is what you will use in the API calls and provider configurations (e.g., `provider=fake-llm`).

> **RECOMMENDED: EXTEND `AbstractLlmClient`**
>
> Instead of implementing `ModelProvider` directly, extend `AbstractLlmClient`. It provides the `applyIfNotNull(value, setter)` helper method for cleanly applying optional configuration parameters from `LlmModelConf`.

Example: `FakeLLMClient.java`

```
package com.uxopian.ai.fakellm.client;

import org.springframework.stereotype.Service;
import com.uxopian.ai.model.llm.connector.AbstractLlmClient;
import com.uxopian.ai.model.llm.connector.LlmModelConf;
import dev.langchain4j.model.chat.ChatModel;
import dev.langchain4j.model.chat.StreamingChatModel;

@Service("fake-llm") // <--- This ID is used in API calls and
provider configurations
public class FakeLLMClient extends AbstractLlmClient {

    @Override
    public ChatModel createChatModelInstance(LlmModelConf params) {
        // Use params to access all configuration: model name, API
key, temperature, etc.
        return new FakeChatModel(params.getModelName());
    }

    @Override
    public StreamingChatModel
createStreamingChatModelInstance(LlmModelConf params) {
        return new FakeStreamingChatModel(params.getModelName());
    }
}
```

The `LlmModelConf` parameter provides access to all configuration fields:

- `params.getModelName()` — The actual model identifier (e.g., `fake-gpt-v1`).
- `params.getApiSecret()` — The API key (decrypted automatically).
- `params.getEndpointUrl()` — The provider API base URL.
- `params.getTemperature()`, `params.getMaxTokens()`, `params.getTimeout()`, etc. — Generation parameters.

These values are the result of merging the provider's global configuration with model-specific overrides. See Parameter Precedence.

Step 5: Register Provider Configuration

After deploying the provider bean, you must create a **provider configuration** (`LlmProviderConf`) to define which models are available and their parameters. This can be done in three ways:

1. **YAML Bootstrapping** — Add a `llm.provider.globals` entry in `llm-clients-config.yml`. It will be loaded into OpenSearch at startup. See Configuration Files.

2. **Admin API** — Create a configuration via `POST /api/v1/admin/llm/provider-conf`. See REST API Reference.

3. **Admin UI** — Use the LLM Provider Management page to visually configure the provider and test connectivity.

Step 6: Packaging & Deployment

Packaging (Fat JAR)

Even for simple connectors, it is best practice to package your provider as a **Fat JAR** (Uber JAR) to ensure all specific dependencies are included and do not conflict with the platform's classpath.

Use the `maven-shade-plugin` or `spring-boot-maven-plugin` (repackage goal).

Deployment (Docker)

Add the JAR to the `/app/provider/` directory in your Docker image.

```
# Start from the specific version of Uxopian-ai
FROM artifactory.arondor.cloud:5001/uxopian-ai:2026.0.0-ft2-full

# Copy your custom provider Fat JAR into the provider directory
COPY ./target/custom-fakellm-provider-1.0.jar /app/provider/
```

Step 7: Verification & Testing

Once your provider is deployed and the application has restarted, you can verify it by sending a standard HTTP request using `CURL`.

The key parameter here is `provider=fake-llm`, which matches the value defined in your `@Service("fake-llm")` annotation.

cURL Example

```
curl -X POST "https://<your-uxopian-endpoint>/api/v1/requests?
provider=fake-llm" \
     -H "Authorization: Bearer <YOUR_ACCESS_TOKEN>" \
     -H "Content-Type: application/json" \
     -d '{
           "inputs": [
             {
               "role": "user",
               "content": [
                 {
                   "type": "text",
                   "value": "Hello, are you a real AI?"
                 }
               ]
             }
           ]
         }'
```

Expected Response

Based on the logic implemented in `FakeChatModel.java`, the system should return a JSON response containing the simulated text:

```
{
  "answer": "FakeLLM (fake-gpt-v1) says: I received your message:
'Hello, are you a real AI?'",
  "inputTokenCount": 10,
  "outputTokenCount": 20,
  "llmName": "fake-gpt-v1",
  "id": "abc12345",
  "createdAt": "2025-05-15T10:00:00"
}
```

Extending Uxopian-ai / Adding a Custom Gateway Authentication Provider

The **Uxopian Gateway (BFF)** is a standalone service, deployed independently from the AI service, that acts as the security entry point for the platform. It is responsible for authenticating incoming requests and establishing a security context (Tenant ID, User ID, Roles) before forwarding traffic to the core AI service.

To integrate with your organization's specific authentication mechanism (e.g., OAuth2 introspection, LDAP, Custom Headers, or JWT validation), you must implement a custom **Auth Provider**.

Overview

The Gateway acts as a proxy. Your custom provider intercepts the `ServerHttpRequest`, validates the credentials, and returns an `AuthenticatedUser` object. The Gateway then injects these details as secure headers (`X-User-Tenant`, `X-User-Id`) when calling the backend.

Prerequisites

- **Java 21+**: The Gateway service requires Java 21 or higher.
- **Maven Settings**: Your `.m2/settings.xml` must be configured to access Uxopian-ai artifacts.

Step 1: Project Configuration

Create a new Maven module. Unlike LLM connectors, this module relies on the `gateway` parent.

`pom.xml` Setup:

You must include the `maven-shade-plugin` configured to exclude Spring Boot starters to avoid conflicts, as this JAR will be loaded dynamically by the main application.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="[http://maven.apache.org/POM/4.0.0]
(http://maven.apache.org/POM/4.0.0)"
         xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance]
(http://www.w3.org/2001/XMLSchema-instance)"
         xsi:schemaLocation="[http://maven.apache.org/POM/4.0.0]
(http://maven.apache.org/POM/4.0.0)
[http://maven.apache.org/xsd/maven-4.0.0.xsd]
(http://maven.apache.org/xsd/maven-4.0.0.xsd)">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>my-custom-auth-provider</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.uxopian.ai.gateway</groupId>
            <artifactId>model</artifactId>
            <version>${project.version}</version>
        </dependency>

        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-test</artifactId>
            <scope>test</scope>
            <version>${reactor.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.5.2</version>
                <executions>
                    <execution>
                        <phase>package</phase>
```

```
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <filters>
                                <filter>
                                    <artifact>*:*</artifact>
                                    <excludes>
                                        <exclude>META-
INF/*.SF</exclude>
                                        <exclude>META-
INF/*.DSA</exclude>
                                        <exclude>META-
INF/*.RSA</exclude>
                                    </excludes>
                                </filter>
                            </filters>
                            <artifactSet>
                                <excludes>

<exclude>org.springframework.boot:*</exclude>
                                    <exclude>org.springframework:*
</exclude>
                                </excludes>
                            </artifactSet>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Step 2: Implement the Provider

You must create a class that implements `com.uxopian.ai.bff.gateway.model.provider.AuthProvider`.

The Gateway uses **Spring WebFlux** (Reactive Stack), so your implementation must return a `Mono<AuthenticatedUser>`.

Crucial: You must annotate this class with `@Service("YourProviderName")`. This name maps directly to the configuration in `application.yml`.

Example: **`DevProvider.java`** *This example demonstrates extracting identity from raw headers (useful for development or behind trusted proxies).*

```
package com.uxopian.ai.bff.gateway.development.provider;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
import com.uxopian.ai.bff.gateway.model.provider.AuthProvider;
import com.uxopian.ai.bff.gateway.model.user.AuthenticatedUser;
import reactor.core.publisher.Mono;

@Service("DevProvider") // <--- This Name is CRITICAL
public class DevProvider implements AuthProvider {

    private static final Logger LOGGER =
LoggerFactory.getLogger(DevProvider.class);
    private static final String HEADER_USER_ID = "X-User-Id";
    private static final String HEADER_USER_ROLES = "X-User-Roles";
    private static final String HEADER_TENANT_ID = "X-User-Tenant";

    @Override
    public Mono<AuthenticatedUser> authenticate(ServerHttpRequest
request) {

        // 1. Extract credentials or headers from the request
        String userId =
request.getHeaders().getFirst(HEADER_USER_ID);
        String rolesRaw =
request.getHeaders().getFirst(HEADER_USER_ROLES);
        String tenantId =
request.getHeaders().getFirst(HEADER_TENANT_ID);

        // 2. Validate (Simplified for example)
        if (!StringUtils.hasText(userId)) {
            LOGGER.warn("DevProvider: Header '{}' is missing.",
HEADER_USER_ID);
            // Return empty user or throw exception depending on
```

```
security requirements
            return Mono.just(new AuthenticatedUser());
        }

        LOGGER.info("Authenticating user '{}'", userId);

        // 3. Construct the AuthenticatedUser object
        AuthenticatedUser user = new AuthenticatedUser();
        user.setId(userId);
        user.setToken("dev-dummy-token"); // Set the token to be
passed downstream

        if (StringUtils.hasText(tenantId)) {
            user.setTenantId(tenantId);
        }

        if (StringUtils.hasText(rolesRaw)) {
            String[] roles = rolesRaw.split(",");
            for (String role : roles) {
                user.addRole(role.trim());
            }
        }

        // 4. Return as a Reactive Mono
        return Mono.just(user);
    }
}
```

Step 3: Gateway Configuration

Once your code is ready, you must configure the Gateway to use this specific provider for the AI service route.

In your `gateway-service` configuration (e.g., `application.yml`), locate the `routes` section. Update the `provider` field to match your `@Service` annotation.

```
app:
  gateway:
    provider-header: X-Provider-ID
  routes:
    - id: uxopian-ai
      uri: http://uxopian-ai-standalone:8080 # The internal address
of the Core Service
      prefix: /uxopian-ai/
      path: /uxopian-ai/**
      rewritePath: /uxopian-ai/?(?<segment>.*), /uxopian-
ai/${segment}

      # MAPPING HAPPENS HERE:
      provider: DevProvider # Must match @Service("DevProvider")

      # Public/Private Path Security Rules
      security:
        - path: /.well-known/**
          public: true
        - path: /swagger-ui/**
          public: true
        - path: /api/v1/admin/**
          # roles: [ "ADMIN" ] # Uncomment to enforce role checks
          public: true
```

Step 4: Deployment

The Gateway loads providers from a plugin directory defined by `auth.provider.path` (default: `provider/`).

Docker Deployment

You must use the **Gateway Service** base image (different from the core AI image).

Dockerfile Example:

```
# Start from the Gateway Service image
FROM artifactory.arondor.cloud:5001/uxopian-ai/gateway-
service:2026.0.0-ft2-full

# Copy your custom Auth Provider Fat JAR into the provider
directory
COPY ./target/my-custom-auth-provider-1.0-SNAPSHOT.jar
/app/provider/
```

Verification

1. **Restart** the Gateway container.
2. **Monitor Logs:** Ensure the service starts without errors regarding missing beans.
3. **Test:** Send a request to the Gateway URL (e.g., `https://gateway/uxopian-ai/api/v1/users/details`) with the headers required by your logic.
4. **Confirm:** Check that the Core Service receives the request with the injected `X-User-Tenant` and `X-User-Id` headers.

Reference / ⚙️ Configuration

This section details the complete configuration of the uxopian-ai service using YAML files.

The framework follows the **Spring Boot externalized configuration** model. Configurations are modular and split into specific files located in the ./config/ directory.

📂 1. General Application Configuration (application.yml)

The entry point for configuration. It defines server settings, security profiles, and imports other configuration modules.

```
app:
  base-url: ${APP_BASE_URL:} # Public URL of the application

server:
  servlet:
    context-path: ${CONTEXT_PATH:} # Context path (e.g., /ai)
  port: ${UXOPIAN_AI_PORT:8080} # Server port

spring:
  config:
    import:
      - "optional:file:./config/llm-clients-config.yml"
      - "optional:file:./config/prompts.yml"
      - "optional:file:./config/goals.yml"
      - "optional:file:./config/mcp-server.yml"
      - "optional:file:./config/opensearch.yml"
      - "optional:file:./config/metrics.yml"
      - "optional:file:./config/application.yml"
  codec:
    max-in-memory-size: 20MB # Adjust for large payloads (e.g.,
images)
  main:
    banner-mode: "off"
  profiles:
    # Use "dev" to disable authentication requirements for testing
    active: ${SPRING_PROFILES_ACTIVE:}

dotenv:
  ignoreIfMissing: true
  filename: .env
```

🔌 2. Enterprise Connectors (FlowerDocs & ARender)

To enable deep integration with your document management ecosystem, specific connectors must be configured in application.yml. These settings allow uxopian-ai to retrieve document content and generate previews.

```
# FlowerDocs Core Integration
ws:
  # URL to the FlowerDocs Core Web Services
  # Examples: https://my-fd/core/ or http://localhost:8081/core
  url: ${FD_WS_URL:#{null}}

# ARender Integration
rendition:
  # Base URL for the ARender Rendition Server
  # Examples: https://my-arender-rendition or http://localhost:8761
  base-url: ${RENDITION_BASE_URL:#{null}}
```

Note: If these URLs are not set (null), the associated features (RAG on FlowerDocs documents, Document Previews) will be disabled.

🛠️ 3. MCP Server Client (mcp-server.yml)

uxopian-ai can act as a client for the **Model Context Protocol (MCP)**, connecting to external MCP servers to access tools or resources via SSE (Server-Sent Events).

```
mcp:
  client:
    name: uxopian-ai-mcp-server
    log-requests: true # Useful for debugging tool calls
  sse:
    # The endpoint of the external MCP server
    url: ${MCP_SSE_URL:http://localhost:8081/uxopian/ai/sse}

  # Logging level for the MCP subsystem
  level:
    org:
      springframework:
        web: TRACE
```

🔗 4. Persistence & Vector Database (opensearch.yml)

Configures the connection to OpenSearch. This is critical for storing:

1. Conversation history
2. Prompts and Goals definitions
3. LLM Provider configurations
4. Vector embeddings (for RAG)
5. Metrics

```
opensearch:
  host: ${OPENSEARCH_HOST:localhost}
  port: ${OPENSEARCH_PORT:9200}
  scheme: ${OPENSEARCH_SCHEME:http}
  username: ${OPENSEARCH_USERNAME:} # Leave empty if no auth
  password: ${OPENSEARCH_PASSWORD:} # Leave empty if no auth

  # CAUTION: Set to true only in development.
  # Forces an index refresh after every write (impacts
performance).
  force-refresh-index: ${OPENSEARCH_FORCE_REFRESH_INDEX:false}
```

📊 5. Metrics & Monitoring (metrics.yml)

Configures **Micrometer** and **Spring Actuator** to export telemetry data directly to OpenSearch. This data powers the Admin Dashboard statistics.

```
management:
  elastic:
    metrics:
      export:
        enabled: true
        # Target OpenSearch/Elasticsearch instance
        host: http://${opensearch.host}:${opensearch.port}
        # Index name for metrics
        index: micrometer-metrics
        auto-create-index: true
  endpoints:
    web:
      exposure:
        include: health,info,loggers
  metrics:
    uxopian-ai:
      enable: true # Enables custom business metrics (Token usage,
ROI, etc.)
    # Standard JVM metrics can be noisy, disable them if not needed
    enable:
      application: false
      tomcat: false
      logback: false
      jvm: false
      system: false
      http: false
      process: false
      disk: false
      executor: false
```

🤖 6. LLM Clients Configuration (llm-clients-config.yml)

This file manages global LLM defaults and dynamic provider configuration.

6.1 Global Defaults & Context

```
llm:
  default:
    provider: ${LLM_DEFAULT_PROVIDER:openai}
    model: ${LLM_DEFAULT_MODEL:gpt-5.1}
    base-prompt: ${LLM_DEFAULT_PROMPT:basePrompt}

  context: ${LLM_CONTEXT_SIZE:10} # Sliding window size (number of
messages)
  debug:
    enabled: ${LLM_DEBUG:false} # Logs full requests/responses
(CAUTION: Sensitive data)
```

6.2 Dynamic Provider Configuration

Since v2026.0.0-ft2, LLM provider configurations are **dynamic entities** stored in OpenSearch. They can be created, updated, and deleted at runtime via the Admin API or the Admin UI.

YAML bootstrapping is still supported: configurations defined in this file are loaded into OpenSearch at startup, then managed dynamically. This section documents the YAML structure used for bootstrapping.

Structure Overview

A provider configuration (`LlmProviderConf`) contains:

- **Provider type** — The name of the registered provider bean (e.g., `openai`, `anthropic`).
- **Default model alias** — Which model to use when no model is specified.
- **Global configuration** (`globalConf`) — Connection and generation parameters shared across all models.

- **Model configurations** (`llModelConfs[]`) — Per-model overrides. Each model inherits from `globalConf` and can override any field.

Configuration Fields

`LlmBaseConf` — Shared by both global and per-model configurations:

Field	Type	Description
`apiSecret`	String	API key or secret. Encrypted at rest via AES-GCM.
`endpointUrl`	String	Provider API base URL.
`temperature`	Double	Sampling temperature (0.0 – 2.0).
`topP`	Double	Nucleus sampling threshold.
`topK`	Integer	Top-K sampling parameter.
`seed`	Integer	Deterministic seed for reproducibility.
`maxTokens`	Integer	Maximum tokens in the response.
`presencePenalty`	Double	Penalizes repeated topics.
`frequencyPenalty`	Double	Penalizes repeated tokens.
`maxRetries`	Integer	Number of retry attempts on failure.
`timeout`	Duration	Request timeout (e.g., `60s`).

Field	Type	Description
`multiModalSupported`	Boolean	Whether the model supports image inputs.
`functionCallSupported`	Boolean	Whether the model supports tool/function calling.
`extras`	Map	Provider-specific key-value pairs (e.g., `deploymentName` for Azure).

`LlmModelConf` — Extends `LlmBaseConf` with two additional fields:

Field	Type	Description
`llmModelConfName`	String	Alias used in prompts and API calls (e.g., `my-gpt5`).
`modelName`	String	Actual model identifier sent to the provider's API (e.g., `gpt-5.1`).

> **CONFIGURATION INHERITANCE**
>
> When processing a request, the service merges global and model-specific settings. Model-level values take precedence over global values. Fields not set at the model level fall back to the global configuration. See Parameter Precedence.

Full YAML Example

```yaml
llm:
  provider:
    # 1. Global Provider Configurations (apply to all tenants)
    globals:
      - provider: openai
        defaultLlmModelConfName: gpt5
        globalConf:
          apiSecret: ${OPENAI_API_KEY:}
          temperature: 0.7
          maxRetries: 3
          timeout: 60s
        llModelConfs:
          - llmModelConfName: gpt5
            modelName: gpt-5.1
            multiModalSupported: true
            functionCallSupported: true
          - llmModelConfName: gpt5-mini
            modelName: gpt-5-mini
            temperature: 0.3
            multiModalSupported: true
            functionCallSupported: true
          - llmModelConfName: gpt4
            modelName: gpt-4.1
            multiModalSupported: true
            functionCallSupported: true

      - provider: anthropic
        defaultLlmModelConfName: claude-sonnet
        globalConf:
          apiSecret: ${ANTHROPIC_API_KEY:}
          endpointUrl: https://api.anthropic.com/v1/
        llModelConfs:
          - llmModelConfName: claude-sonnet
            modelName: claude-sonnet-4-20250514
            multiModalSupported: true
            functionCallSupported: true
```

```
    # 2. Tenant-Specific Overrides
    tenants:
      - tenantId: tenant-A
        mergeStrategy: MERGE  # Options: MERGE, OVERWRITE,
CREATE_IF_MISSING
        providers:
          - provider: openai
            defaultLlmModelConfName: gpt5
            globalConf:
              apiSecret: sk-tenant-a-specific-key
            llModelConfs:
              - llmModelConfName: gpt5
                modelName: gpt-5.1
                multiModalSupported: true
                functionCallSupported: true
```

Merge Strategies

When tenant-specific configurations are defined, the `mergeStrategy` field controls how they combine with global configurations:

Strategy	Behavior
`MERGE`	Tenant providers are merged with globals. Matching providers are updated; non-matching ones are added.
`OVERWRITE`	Tenant configuration completely replaces the global configuration.
`CREATE_IF_MISSING`	Tenant providers are only added if no global configuration exists for that provider type.

API Secret Encryption

API secrets stored in OpenSearch are encrypted using **AES-GCM**. Set the encryption key via the `APP_SECURITY_SECRET_KEY` environment variable (Base64-encoded AES key). See Environment Variables.

🧩 7. Bootstrapping Prompts, Goals & LLM Providers

These files are used to initialize the system with default data. They support global definitions and tenant-specific overrides. Data defined in YAML is loaded into OpenSearch at startup and can then be managed dynamically via the Admin API or UI.

Prompts (prompts.yml)

```
prompts:
  backup:
    path: ${PROMPTS_BACKUP_PATH:./prompts/}

  # 1. Global Prompts (Apply to everyone)
  globals:
    - id: basePrompt
      role: SYSTEM
      content: |
        You are Xopia. You were born in 2025...
      reasoningDisabled: false
      requiresMultiModalModel: false
      requiresFunctionCallingModel: false

  # 2. Tenant Specifics
  tenants:
    - tenantId: tenant-A
      mergeStrategy: merge # Options: merge, override,
createIfMissing
      prompts:
        - id: specificPrompt
          role: USER
          content: "Use only the information explicitly present in
the document..."
          requiresMultiModalModel: false
          requiresFunctionCallingModel: false
```

Goals (goals.yml)

```yaml
goals:
  backup:
    path: ${GOALS_BACKUP_PATH:./goals/}

  globals: []

  tenants:
    - tenantId: tenant-A
      mergeStrategy: merge
      goalGroups:
        - id: compare
          goals:
            - promptId: detailedComparisonForTenantA
              filter: "[[${documentType == 'contract'}]]"
              index: 125
            - promptId: genericComparison
              filter: "true"
              index: 1000
```

LLM Providers (llm-clients-config.yml)

LLM Provider configurations follow the same bootstrapping pattern. They are defined under `llm.provider.globals` and `llm.provider.tenants` in `llm-clients-config.yml`. See Section 6.2 for the full YAML structure and field reference.

For a quick reference of all environment variables available for Docker deployments, see Environment Variables.

Reference / REST API Reference

All interactions with uxopian-ai go through its REST API. This page provides a complete overview of the endpoints and how to access the interactive documentation.

For practical usage examples with cURL and JavaScript, see Using the REST API.

Interactive Documentation (Swagger UI)

The service exposes a full **OpenAPI 3.1.0** specification. We highly recommend using the built-in Swagger UI for exploring the API, inspecting schemas, and testing requests in real-time.

Accessing Swagger UI

Depending on your environment, the Swagger UI is available at:

- **Local / Docker**: `http://localhost:8080/swagger-ui.html` (Default port is 8080)

Note: If a custom `CONTEXT_PATH` is configured in `application.yml`, append it to the base URL (e.g., `http://localhost:8080/ai/swagger-ui.html`).

PUBLIC ACCESS

The Swagger UI is publicly accessible (no authentication required). Endpoints are organized by tags prefixed with `Admin -` for admin operations, making it easy to explore the full API surface.

Authentication

The API expects authentication via headers injected by your Gateway or BFF. See Security Model for the full explanation.

Header	Required	Description
`X-User-TenantId`	Yes	Tenant isolation key.
`X-User-Id`	Yes	Unique User ID.
`X-User-Roles`	No	Comma-separated roles (e.g., `admin`).
`X-User-Token`	No	Original user token for downstream context.

API Endpoints

Conversations

Manage the lifecycle of chat sessions.

Method	Endpoint	Description
`POST`	`/api/v1/conversations`	Create a new conversation.
`GET`	`/api/v1/conversations`	List user conversations (paginated).
`GET`	`/api/v1/conversations/{id}`	Retrieve full details of a conversation.
`DELETE`	`/api/v1/conversations/{id}`	Delete a conversation.

Requests (Chat)

Send messages and interact with the LLM.

Method	Endpoint	Description
`POST`	`/api/v1/requests`	Send a message and get a synchronous response.
`POST`	`/api/v1/requests/stream`	Send a message and receive the response as an SSE stream.
`POST`	`/api/v1/requests/retry`	Regenerate the last answer.

Query Parameters:

- `conversation` (Required) — The conversation ID.

- `provider` (Optional) — Override the LLM provider for this request.
- `model` (Optional) — Override the LLM model for this request.

Administration — Prompts

Endpoints restricted to users with the `admin` role.

Method	Endpoint	Description
`POST`	`/api/v1/admin/prompts`	Create a new prompt.
`PUT`	`/api/v1/admin/prompts`	Update an existing prompt.
`GET`	`/api/v1/admin/prompts/{id}`	Get a specific prompt by ID.
`DELETE`	`/api/v1/admin/prompts/{id}`	Delete a prompt.

Administration — Goals

Method	Endpoint	Description
`POST`	`/api/v1/admin/goals`	Create a new goal.
`PUT`	`/api/v1/admin/goals`	Update an existing goal.
`GET`	`/api/v1/admin/goals`	Get all goals.
`DELETE`	`/api/v1/admin/goals/{id}`	Delete a goal.

Administration — LLM Providers

Manage LLM provider configurations at runtime. See LLM Provider Management for the full UI guide.

Method	Endpoint	Description
`GET`	`/api/v1/admin/llm/providers`	List all registered provider types (bean names).
`GET`	`/api/v1/admin/llm/provider-conf`	List all provider configurations for the tenant.
`GET`	`/api/v1/admin/llm/provider-conf/{id}`	Get a specific provider configuration by ID.
`POST`	`/api/v1/admin/llm/provider-conf`	Create a new provider configuration.
`PUT`	`/api/v1/admin/llm/provider-conf/{id}`	Update an existing provider configuration.
`DELETE`	`/api/v1/admin/llm/provider-conf/{id}`	Delete a provider configuration.

Administration — Statistics

All statistics endpoints accept an optional `interval` query parameter to control time-series granularity. Supported values: `HOUR`, `DAY`, `WEEK`, `MONTH`, `YEAR`. Default: `DAY`.

Method	Endpoint	Description
`GET`	`/api/v1/admin/stats/global`	Aggregate counters (total requests, tokens, time saved).
`GET`	`/api/v1/admin/stats/timeseries?interval=DAY`	Time-series activity data (requests, tokens over time).
`GET`	`/api/v1/admin/stats/llm-distribution`	Model usage distribution breakdown.
`GET`	`/api/v1/admin/stats/top-prompts-time-saved`	Top prompts ranked by estimated time saved.
`GET`	`/api/v1/admin/stats/feature-adoption`	Advanced feature adoption rates (multi-modal, function calling).

Reference / Environment Variables Reference

For Docker and production deployments, override these variables instead of editing YAML files directly. Set them in your `docker-compose.yml`, `.env` file, or container orchestrator.

General

Variable	Description	Default
`SPRING_PROFILES_ACTIVE`	Active profile (use `dev` to disable auth).	*empty*
`UXOPIAN_AI_PORT`	Application server port.	`8080`
`APP_BASE_URL`	Public base URL of the service.	*empty*
`CONTEXT_PATH`	Servlet context path (e.g., `/ai`).	*empty*

Security

Variable	Description	Default
`APP_SECURITY_SECRET_KEY`	Base64-encoded AES key for encrypting LLM provider API secrets at rest.	*empty*

LLM Providers

Variable	Description	Default
`LLM_DEFAULT_PROVIDER`	Default LLM provider.	`openai`
`LLM_DEFAULT_MODEL`	Default LLM model name.	`gpt-5.1`
`LLM_DEFAULT_PROMPT`	Default base prompt ID.	`basePrompt`
`LLM_CONTEXT_SIZE`	Sliding window size (messages).	`10`
`LLM_DEBUG`	Log full LLM requests/responses (sensitive).	`false`
`OPENAI_API_KEY`	API Key for OpenAI.	*empty*
`ANTHROPIC_API_KEY`	API Key for Anthropic.	`none`
`AZURE_OPENAI_API_KEY`	API Key for Azure OpenAI.	`none`
`GEMINI_API_KEY`	API Key for Google Gemini.	`none`

Variable	Description	Default
`MISTRAL_API_KEY`	API Key for Mistral AI.	`none`
`HUGGINGFACE_API_KEY`	API Key for HuggingFace.	`none`
`BEDROCK_AWS_ACCESS_KEY`	AWS access key for Bedrock.	`none`
`BEDROCK_AWS_SECRET_KEY`	AWS secret key for Bedrock.	`none`

> **DYNAMIC PROVIDER CONFIGURATION**
>
> Since v2026.0.0-ft2, LLM API keys are primarily managed via the dynamic provider configuration stored in OpenSearch. The individual environment variables above (`OPENAI_API_KEY`, etc.) remain functional for bean initialization and YAML bootstrapping.

OpenSearch

Variable	Description	Default
`OPENSEARCH_HOST`	Hostname of the OpenSearch instance.	`localhost`
`OPENSEARCH_PORT`	Port of the OpenSearch instance.	`9200`
`OPENSEARCH_SCHEME`	Connection scheme (`http` or `https`).	`http`

Variable	Description	Default
`OPENSEARCH_USERNAME`	OpenSearch username (if secure).	*empty*
`OPENSEARCH_PASSWORD`	OpenSearch password (if secure).	*empty*
`OPENSEARCH_FORCE_REFRESH_INDEX`	Force index refresh after writes (dev only, impacts perf).	`false`

Integrations

Variable	Description	Default
`FD_WS_URL`	FlowerDocs Core Web Services URL.	*null*
`RENDITION_BASE_URL`	ARender Rendition Server base URL.	*null*

Variable	Description	Default
`MCP_SSE_URL`	MCP server SSE endpoint URL.	`http://localhost:8081/uxopian/ai/s`

Backup

Variable	Description	Default
`PROMPTS_BACKUP_PATH`	Path to load/store prompt backups.	`./prompts/`
`GOALS_BACKUP_PATH`	Path to load/store goal backups.	`./goals/`

Admin Panel / Admin Dashboard

The **Uxopian-ai Admin Panel** acts as the central command center for your AI infrastructure. It provides a visual interface to manage resources, monitor usage statistics, and oversee user activity.

> **ACCESS**
>
> The Admin Panel is restricted to users with the **admin role**. **URL:** `https://<your-uxopian-endpoint>/admin`

Key Features

The dashboard offers quick access to the primary management modules:

- **Prompts:** Manage, test, and analyze AI prompt configurations.
- **LLM Providers:** Configure and test LLM provider connections.
- **Statistics:** View global health metrics, usage trends, and ROI data.
- **Users:** Monitor user activity and view detailed interaction history.

Resources

The dashboard also provides direct access to developer tools:

- **Official Documentation:** Link to these guides.

- **Swagger UI:** Access the interactive API documentation for exploring and testing endpoints directly.

Admin Panel / Prompts Management

Prompts are the building blocks of your AI interactions. This section allows administrators to manage the lifecycle of these prompts, from creation to performance analysis.

1. Prompt List & Search

The main view displays all available prompts for the current tenant.

- **Search:** Find specific prompts by their ID or content.
- **Actions:** Create new prompts, edit existing ones, or view their specific statistics.
- **API Reference:** `GET /api/v1/admin/prompts`

2. Prompt Editor (CRUD)

The editor provides granular control over prompt behavior.

Content & Configuration

- **Content Editor:** Edit the **Thymeleaf template** used to generate the prompt dynamically. See The Templating Engine for syntax details.
- **Model Configuration:** Set the default LLM provider (e.g., `openai`) and model (e.g., `gpt-5.1`). See Choosing the Right Model for guidance.
- **Capabilities:** Toggle flags based on the prompt's needs:

- **Reasoning:** Enable or disable reasoning capabilities.
- **Multi-modal:** Flag if the prompt requires image inputs.
- **Function Calling:** Flag if the prompt triggers external tools.

ROI Settings

- **Time Saved:** Define an estimated "Time Saved per usage" (in seconds). This value is used to calculate the Return on Investment (ROI) metrics across the platform.

3. Prompt Statistics

Each prompt has a dedicated statistics view with performance metrics:

- **Usage Count:** The total number of times this prompt has been triggered.
- **Feedback:** A breakdown of user ratings (**Good**, **Bad**, **Neutral**) to assess output quality.
- **Cost & ROI:** Token consumption (`totalCost`, `costAverage`) and the specific time saved by this prompt.

4. Prompt Tester

The **Prompt Tester** allows you to execute a prompt directly from the admin interface — without needing to set up a conversation or use external tools like cURL.

How It Works

1. **Variable Detection:** When you open the tester for a prompt, it automatically parses the Thymeleaf template and detects all variables (e.g.,

`${documentId}`, `${language}`).

2. **Variable Configuration:** For each detected variable, an input field is displayed. You can provide:
   - **Text values** — For string variables like document IDs, language codes, or user queries.
   - **Image values** — For multi-modal prompts that expect Base64-encoded images.
3. **Execute:** Click "Test" to send the prompt with the configured variables to the LLM. The system uses the prompt's `defaultLlmProvider` and `defaultLlmModel` settings.
4. **Results:** The tester displays:
   - The LLM response.
   - Token usage (input/output).
   - Response latency.
5. **cURL Generation:** The tester generates the equivalent **cURL command** for the test configuration, making it easy to reproduce the request from a terminal or integrate into scripts.

> **ITERATE QUICKLY**
>
> Use the Prompt Tester to refine your prompt content and model selection before deploying to production. Adjust the template, change variables, and re-test — all without leaving the admin panel.

Admin Panel / LLM Provider Management

The **LLM Provider Management** page allows administrators to configure, test, and manage LLM provider connections at runtime — without restarting the service.

Each provider configuration defines connection parameters (API key, endpoint, timeouts) and a list of available models with their specific overrides.

---

1. Provider List

The main view displays all configured LLM providers for the current tenant.

- **Search & Filter**: Quickly find providers by name or type.
- **Actions**: Create, edit, or delete provider configurations.
- **Status Indicators**: See at a glance which providers have valid configurations.
- **API Reference**: `GET /api/v1/admin/llm/provider-conf`

Each row shows the provider type (e.g., `openai`, `anthropic`), the default model alias, and the number of configured models.

---

2. Provider Editor

The editor provides granular control over a provider's configuration. It is divided into three sections.

Provider Identity

Field	Description
Provider	The provider type (must match a registered Spring bean: `openai`, `anthropic`, `azure`, `gemini`, `mistral`, `ollama`, etc.)
Default Model Conf Name	The alias of the model to use by default when no model is specified in the request.

Global Configuration

These settings apply to **all models** under this provider unless overridden at the model level.

Field	Type	Description
`apiSecret`	String	API key or secret for authentication. Encrypted at rest (AES-GCM).
`endpointUrl`	String	Base URL for the provider's API.
`temperature`	Double	Sampling temperature (0.0 – 2.0).
`topP`	Double	Nucleus sampling threshold.

Field	Type	Description
`topK`	Integer	Top-K sampling parameter.
`seed`	Integer	Deterministic seed for reproducibility.
`maxTokens`	Integer	Maximum tokens in the response.
`presencePenalty`	Double	Penalizes repeated topics.
`frequencyPenalty`	Double	Penalizes repeated tokens.
`maxRetries`	Integer	Number of retry attempts on failure.
`timeout`	Duration	Request timeout (e.g., `60s`, `PT2M`).
`multiModalSupported`	Boolean	Whether the provider supports image inputs.
`functionCallSupported`	Boolean	Whether the provider supports tool/function calling.
`extras`	Map	Provider-specific key-value pairs (e.g., `deploymentName` for Azure).

Model Configurations

Each model entry represents a specific model alias available through this provider. Models **inherit** all global configuration values and can override any of them.

Field	Description
Model Conf Name (`llmModelConfName`)	The alias used in prompts and API calls (e.g., `my-gpt5`).
Model Name (`modelName`)	The actual model identifier sent to the provider's API (e.g., `gpt-5.1`).
(all global fields)	Any field from the global configuration can be overridden per model.

> **CONFIGURATION INHERITANCE**
>
> When processing a request, the service **merges** global and model-specific settings. Model-level values take precedence. For example, if the global `temperature` is `0.7` but a specific model sets `temperature: 0.2`, the model-level value (`0.2`) is used. Fields not overridden at the model level fall back to the global value.

---

3. Provider Detail

Selecting a provider from the list opens the detail view, which combines:

- **Editor Tab** — The full provider editor (see above).
- **Tester Tab** — The connection tester (see below).

---

4. Connection Tester

The Connection Tester lets you verify that a provider configuration is working correctly **before** using it in production prompts.

How It Works

1. Select a **model** from the provider's configured model list.
2. Click **Test Connection**.
3. The system sends a minimal test request to the provider's API.
4. Results are displayed with:
    - **Status badge**: Success or failure indicator.
    - **Response details**: Model response, latency, and token usage.
    - **Error details**: If the test fails, the full error message is shown for debugging.

> **TEST AFTER CHANGES**
>
> After modifying API keys, endpoints, or model names, always use the Connection Tester before saving. This prevents misconfigured providers from affecting live prompts.

---

5. Per-Tenant Configuration

LLM provider configurations support **multi-tenancy**. Configurations can be defined at the global level and then customized per tenant using a **merge strategy**:

Strategy	Behavior
`MERGE`	Tenant-specific providers are merged with global ones. Matching providers are updated; non-matching ones are added.
`OVERWRITE`	Tenant configuration completely replaces the global configuration.
`CREATE_IF_MISSING`	Tenant-specific providers are only added if no global configuration exists for that provider.

This allows a central admin to define a base set of providers (e.g., OpenAI with a shared API key) while individual tenants can add their own providers or override API keys.

API Reference

All operations require the `admin` role (`X-User-Roles: admin`).

Method	Endpoint	Description
`GET`	`/api/v1/admin/llm/providers`	List all registered provider types (bean names).
`GET`	`/api/v1/admin/llm/provider-conf`	List all provider configurations for the tenant.

Method	Endpoint	Description
`GET`	`/api/v1/admin/llm/provider-conf/{id}`	Get a specific provider configuration by ID.
`POST`	`/api/v1/admin/llm/provider-conf`	Create a new provider configuration.
`PUT`	`/api/v1/admin/llm/provider-conf/{id}`	Update an existing provider configuration.
`DELETE`	`/api/v1/admin/llm/provider-conf/{id}`	Delete a provider configuration.

Example: Creating a Provider Configuration

```
curl -X POST "http://localhost:8080/api/v1/admin/llm/provider-conf"
\
  -H "Content-Type: application/json" \
  -H "X-User-TenantId: enterprise-corp-a" \
  -H "X-User-Roles: admin" \
  -d '{
    "provider": "openai",
    "defaultLlmModelConfName": "gpt5",
    "globalConf": {
      "apiSecret": "sk-your-api-key-here",
      "temperature": 0.7,
      "maxRetries": 3,
      "timeout": "60s"
    },
    "llModelConfs": [
      {
        "llmModelConfName": "gpt5",
        "modelName": "gpt-5.1",
        "multiModalSupported": true,
        "functionCallSupported": true
      },
      {
        "llmModelConfName": "gpt5-mini",
        "modelName": "gpt-5-mini",
        "temperature": 0.3,
        "multiModalSupported": true,
        "functionCallSupported": true
      }
    ]
  }'
```

Related Resources

- Configuration Files — Dynamic Provider Configuration — YAML bootstrapping reference.
- Core Concepts — Parameter Precedence — How model parameters are resolved.
- Adding a New LLM Provider — Creating custom provider connectors.

Admin Panel / Global Statistics

The **Statistics** page provides a high-level overview of your system's health, usage trends, and Return on Investment (ROI).

Time Interval Selector

All time-based views support a configurable **time interval**. Use the interval selector to adjust the granularity of the data:

Interval	Description
`HOUR`	Hourly breakdown (useful for monitoring recent activity).
`DAY`	Daily aggregation (default).
`WEEK`	Weekly summary.
`MONTH`	Monthly overview.
`YEAR`	Yearly totals.

The selected interval applies to the Usage Trends view and is passed as the `interval` query parameter when calling the statistics API.

Global Metrics

Real-time counters provide an instant view of the organization's total consumption:

- **Total Requests:** The aggregate number of interactions processed.
- **Total Conversations:** The count of unique conversation threads.
- **Total Tokens:** The sum of all Input and Output tokens consumed.
- **Total Time Saved:** The cumulative estimated hours saved, calculated based on the ROI settings of utilized prompts.

API: `GET /api/v1/admin/stats/global`

Usage Trends

Time-series charts displaying activity over the selected time interval:

- **Request Volume:** Number of requests over time.
- **Token Consumption:** Input and output tokens over time.

API: `GET /api/v1/admin/stats/timeseries?interval=DAY`

LLM Distribution

A breakdown of which AI models and providers are being utilized the most. Helps identify model concentration and optimize costs.

API: `GET /api/v1/admin/stats/llm-distribution`

Top Prompts by Time Saved

A ranking of the most valuable prompts in terms of estimated productivity gains. Each prompt's `timeSaved` value (set during prompt creation) is multiplied by its usage count to calculate the total ROI.

API: `GET /api/v1/admin/stats/top-prompts-time-saved`

Feature Adoption

Visualizes the usage rate of advanced capabilities:

- **Multi-Modal:** Percentage of requests that include image inputs.
- **Function Calling:** Percentage of requests that trigger external tools.

API: `GET /api/v1/admin/stats/feature-adoption`

Admin Panel / User Management

The User Management section allows administrators to monitor activity at the user level to understand adoption patterns and usage intensity.

User List

The main view displays all users who have interacted with the system, with the following metrics per user:

Metric	Description
Conversation Count	The number of sessions created by the user.
Token Usage	The total Input and Output tokens consumed.
Request Count	The total number of individual interactions.

User Details

Selecting a user opens a detailed view of their activity.

- **Summary Stats:** Recaps the user's total consumption and activity metrics.
- **Conversation History:** Browse the full list of conversations created by the user.

- **Request History:** Inspect individual requests and their LLM responses. This is useful for support troubleshooting and auditing purposes.

Frequently Asked Questions

Can Uxopian AI plug into other ECMs, viewers, and LLM providers? Can I buy Uxopian AI alone?

Yes. Uxopian AI can be deployed independently from FlowerDocs and ARender, and connected to third-party ECM platforms (e.g., OpenText) and third-party viewers (e.g., OpenText Intelligent Viewing). It can also be configured to use LLM providers other than OpenAI (from the list of providers already supported by Uxopian AI).

The main condition is integration: a system integrator or partner must implement the connector(s) for the target ECM/viewer/application. Once implemented, these connectors are reusable across projects.

How it works (prompt templating + “prompt helpers”)

Uxopian AI sends requests to LLMs through a prompt templating system. Prompts can include expressions that are evaluated at runtime to fetch context from the surrounding application or content system.

Documentation: Creating Custom Helpers

Example (as used in standard demos), where Uxopian AI summarizes the document currently opened by the user:

```
Summarize the following document. It has to stay in less than 60
words, but have the key information to grasp the bulk of the
conversation. You can use markdown, only to put in bold the
critical pieces.

Document content:
[[${documentService.extractTextualContent(documentId)}]]
```

When the prompt templating engine encounters the expression inside `[[ ... ]]`, it calls a backend helper to resolve it. In the example above, `documentService.extractTextualContent(documentId)` retrieves the full text of the document (using the document ID already available because the user opened it in the UI). The final prompt sent to the LLM is the original template plus the resolved document content.

You can verify what is actually sent to the LLM by reviewing the history of LLM exchanges in the Users section of Uxopian AI.

Plugging into other ECMs/viewers

The `documentService.extractTextualContent(...)` behavior in the example is provided through an integration component called a **prompt helper**. The helper used in the demo is specific to ARender, but the mechanism is designed to be pluggable.

That means you can implement a new prompt helper for another system. For example:

- A Documentum helper that retrieves full text given a Documentum document ID
- An OpenText helper that retrieves full text and metadata from OpenText

- A “case context” helper for Salesforce that injects the current case details (customer, policy, claim status, next actions, etc.)

How to create custom prompt helpers: Creating Custom Helpers

Uxopian AI also includes existing helpers (for example, a helper for FlowerDocs to access documents by FlowerDocs ID), which can be used as a reference.

Integrating the assistant UI into other applications

On the UI side, the assistant is packaged as a **web component**, which is a highly portable format for embedding UI elements in web applications. This makes it straightforward to integrate Uxopian AI into any web app that provides an extension mechanism (plugin areas, custom widgets, embedded panels, etc.).

Documentation: Embedding in a Web Page

Summary

- Uxopian AI can be purchased and deployed on its own.
- Integration with third-party ECM/viewers/apps is achieved by implementing reusable connectors (prompt helpers and UI embedding).
- Switching LLM providers is supported (within the list of providers supported by Uxopian AI).
- Most customers rely on a system integrator/partner to implement the connectors once, then reuse them across deployments.

Can I use Xopia to trigger AI agents, instead of a "flat" LLM-based conversation.

The term “agent” is still understood in different ways. Uxopian AI will allow later in 2026 to declare new agents. when it comes to integrating existing ones, let's separate two use cases:

1) Conversational agents (chat-based agent services).
If your “agent” exposes a messaging / chat API (and runs its own reasoning loop), you can use the Uxopian Assistant UI to converse with it by implementing a **provider / model class** that connects to that agent, the same way a provider/model would connect to an LLM endpoint. From the UI standpoint, it remains a normal assistant conversation; under the hood, the “model” is your agent service. See the How to Guides section for this.

2) Action agents (agents that execute tasks).
If your “agent” is something you call to perform an action (redaction, metadata updates, lookups, update customer in CRM, etc.) rather than a chat endpoint, then the right concept is a **Tool / Function Calling** integration. Uxopian AI can invoke declared tools behind the scenes when the user asks for actions (e.g., “Redact these documents and remove addresses”).
See: Creating Custom Tools
We also plan to add a direct MCP bridge later this year.

Can I connect customer-developed LLM endpoints (or proprietary AI services) in Xopia?

Yes. Customer-developed LLM endpoints (or proprietary AI services) can be integrated by implementing a **provider / model connector** so that Xopia routes calls to that endpoint. This can be used either as a classical LLM provider, or to connect to a conversational-agent service (chat/messaging API) presented as a “model” to the Assistant UI.

Do we have an internal paper about “agents” in Uxopian AI?

Not at the moment as a single dedicated paper. The most authoritative references today are the extension documentation (custom tools, and provider/model extensions). We will integrate the concept of agent later in 2026 in Uxopian AI framework.