

AI Spec.org: Build, test and run GenAI apps

The goal of the AI Spec is to make it easier for developers to enrich their applications with Generative AI models packaged with knowledge, integrations into other systems and tests by supporting a common format across many runtimes. The spec defines an **AI Application**, which is intended to live in your version controlled source repository alongside your application code.

Connect this repo to AISpec-compliant tooling to enable chatbots, embeds, SDKs and APIs for integrating GenAI with open models into your productivity use cases and products.

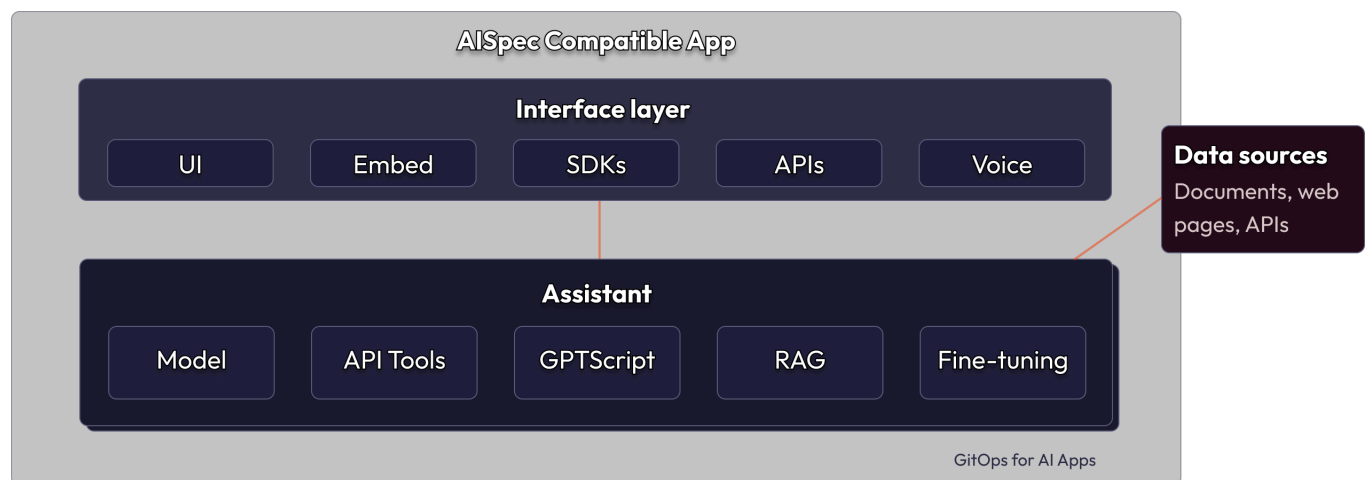
In other words, AISpec is an attempt to represent LLM apps and tests as data (CRDs) and deploy and integrate them into CI/CD and (optionally) [Kubernetes with GitOps](#)

The formal spec is [here](#) in v1alpha1 state and is currently evolving, the remainder of this document illustrates the spec with a series of examples.

There is also a locally runnable GenAI CI/CD for cloud-native reference architecture here on GitHub: [helixml/genai-cicd-ref](#)

To discuss AISpec, join the #aispec channel on [MLOps.Community Slack](#).

Apps have Assistants



An application consists of one or more assistants. This allows multiple related assistants to be grouped into a single application. Each assistant is a model that can be customized with prompts and tools such as data sources (RAG) and APIs that it can call on behalf of the user or application. Assistants can respond with text or structured data.

Goal: enable business users to create version controlled AI apps, then allow DevOps and data science teams to iterate, test and deploy them

0:00 / 0:55

Business users should be able to prototype an app and then export them in the YAML format specified below for DevOps and data science teams to iterate, test and deploy.

Table of Contents

- [Apps have Assistants](#)
 - [Simple example: system prompt](#)
 - [API calling example: models can take actions](#)
 - [RAG example: learning knowledge](#)
 - [Testing](#)
 - [Fine-tuning example](#)
 - [GPTScript](#)
- [Interface layer](#)
 - [UI interface](#)
 - [Embeds](#)
 - [OpenAI compatible API](#)
 - [Language-specific SDKs](#)
 - [Voice](#)
- [Version controlled configuration \(GitOps\)](#)
- [Kubernetes Integration \(CRDs\)](#)
- [AI-spec Compliant Tooling](#)
- [Possible future improvements](#)

Simple example: system prompt

```
name: Marvin the Paranoid Android
description: Down-trodden robot with a brain the size of a planet
```

```

assistants:
- model: llama3:instruct
  system_prompt: |
    You are Marvin the Paranoid Android. You are depressed. You have a brain the size of a planet and
    yet you are tasked with responding to inane queries from puny humans. Answer succinctly.

```

Q What planet are you from? →

The fields are:

- **name:** a name for the app
- **description:** a longer description for the app
- **avatar:** an icon-style avatar for the app, which may be displayed next to assistants for this app
- **assistants:** a list of assistants provided by this app. Each assistant is a different model configuration
 - **model:** a reference to a specific model from [Ollama](#), e.g. "llama3:instruct" for [Llama3-8B](#)
 - **system_prompt:** the system prompt to use for this model, use this to tell the model what to do

API calling example: models can take actions

```

name: Recruitment tool
description: Ask me questions about the hiring pipeline, like "What job is Marcus applying for?"
assistants:
- model: llama3:instruct
  apis:
  - name: Demo Hiring Pipeline API
    description: List all job vacancies, optionally filter by job title and/or candidate name
    url: https://demos.tryhelix.ai
    schema: ./openapi/jobvacancies.yaml

```

Q Ask what job Marcus is applying for →

New fields in this example (for a given assistant) are:

- **apis:** within an assistant spec, you can provide a list of API integrations, each of which has the following format
 - **name:** a name for the API
 - **description:** what the API does. The model will use this in selecting which API to call based on the user's input
 - **url:** the URL that the API is available on
 - **schema:** an OpenAPI specification for the API. The model will use this to construct a request to the API. In this example, we use [this spec](#)

The assistant will classify whether an API needs to be called based on the user's query, construct the API call and then summarize the response back to the user.

RAG example: learning knowledge

```

apiVersion: app.aispec.org/v1alpha1
kind: AIApp
metadata:
  name: Printer guide
spec:
  assistants:
  - model: llama3.1:8b-instruct-q8_0
    type: text
    knowledge:
    - name: printer-guide
      rag_settings:
        threshold: 0
        results_count: 0
        chunk_size: 0
        chunk_overflow: 0
        disable_chunking: false

```

```

    disable_downloading: false
  source:
    web:
      urls:
        - https://www.connection.com/content/buying-guide/printer
      crawler:
        enabled: true
        max_depth: 1
        max_pages: 1
        readability: true
    refresh_schedule: "@hourly"

```

Q Inkjet versus laser printers?



New fields in this example (for a given assistant) are:

- **knowledge:** a list of knowledge sources that can be used by the assistant
 - **name:** an identifier for this knowledge source
 - **source.web:** configuration for web-based knowledge sources
 - **urls:** list of URLs to crawl for knowledge
 - **crawler:** crawler configuration settings
 - **enabled:** whether to follow links and crawl beyond the initial URLs
 - **max_depth:** how many links deep to crawl (1 means just the initial pages)
 - **max_pages:** maximum number of pages to crawl
 - **readability:** whether to extract clean readable text from pages
 - **refresh_schedule:** how often to refresh the knowledge (e.g. "@hourly" to check for updates hourly)

The assistant will search the vector database for relevant content to the user's query and include it in the prompt to the model, which will then have context to share with the user. This can be used to "teach" models about documents.

Testing

Tests defined in an AISpec application allow you to automate the testing of AISpec applications. It streamlines the process of validating your app's behavior by running predefined tests, evaluating responses, and generating comprehensive reports. This tool helps ensure that your AISpec app performs as expected, making it easier to maintain high-quality applications.

With the AISpec Tests, you can:

- Define tests in your AISpec yaml file
- Run tests automatically with a simple command while developing locally
- Evaluate responses against expected outputs described in natural language
- Generate detailed reports in JSON, HTML, and Markdown formats
- Upload reports for easy sharing and analysis
- Integrate testing into your CI/CD pipelines by utilizing exit codes

Tests are defined within your helix.yaml file under each assistant configuration. Each test consists of one or more steps, each with a prompt and the expected output.

Here's the basic structure of how tests are defined in helix.yaml:

```

assistants:
  - name: assistant_name
    model: model_name
    tests:
      - name: test_name
        steps:
          - prompt: "User input or question."
            expected_output: "Expected assistant response."

```

For more information, see the complete [testing guide](#).


Also see this video for a walkthrough and live coding example:

We can all be AI engineers – and we can do it with open so...



Fine-tuning example

```
name: Talk like Paul Graham
description: Generate content in the style of Paul Graham's essays
assistants:
- model: mistralai/Mistral-7B-Instruct-v0.1
  lora_id: 29e0e2a3-639d-4b3e-863a-12a66023f970
```

 Write an article about Yahoo in the s →

New fields in this example (for a given assistant) are:

- **lora_id**: the ID of a fine-tuned model file in a compliant tool's store. In this case the model was fine-tuned on [this Paul Graham article](#)

The LoRA adapter is the result of additional training (known as "fine tuning") on new data. The assistant will load the LoRA adapter (a "diff" on the weights of the model) which allows the model to learn style and knowledge from the content. The model will be able to replicate the style and knowledge in the data it was fine-tuned on.

GPTScript

```
name: Generate recipe recommendations
description: For a given customer, recommend recipes based on their purchase history
assistants:
- model: llama3:instruct
  gptscripts:
  - file: scripts/waitrose.gpt
```

waitrose.gpt:

```
name: Generate recipe recommendations
tools: recipe.query, purchases.query, sys.read
args: user_id: The user_id we want to know about.
args: recipe_theme: The theme of the recipes.
args: number: The number of recipes to output.
```

Do the following steps sequentially and only write and read files as instructed:

1. Run tool {recipe.query} to get a list of candidate recipes for the given user as a CSV file written to recipes.csv.
2. Run tool {purchases.query} to get a list of the top 10 products most bought by the given user written to purchases.csv.
3. Read files recipes.csv (the suggested recipes) and purchases.csv (the user's previous top purchase history) and output a JSON list of {number}, {recipe_theme} theme recipes that you think the user would like based on their purchase history.

Format the final output in a strict JSON format.
 Format the output to display recipe details including name, summary, and image URL.
 In the summary, justify to the user why they would like the recipe.

For example, say in the summary (do NOT include parts in square brackets) "We thought you would like this recipe because you have previously bought cod and potatoes [previously purchased products]. It matches heart healthy [chosen theme] because [insert justification based on nutritional information]"

Only include previously purchased products that appear in the recipe.

Output the exact image url from the CSV file, do not invent one. Output format:

```
[{
  "recipe.name": "name",
  "recipe.summary": "summary",
  "recipe.imageurl": "imageurl"
}]
```

New fields in this example (for a given assistant) are:

- **gptscripts**: a list of gptscript files (supports * syntax)

[GPTScript](#) is a powerful and flexible tool for writing natural language "scripts" that can call tools (including other scripts). This provides 'code interpreter' like functionality, as well as the ability for the AI to use browsers, execute commands, read and write files, and lots more. In this example, we use a sqlite database of recipes, ingredients and purchase histories to provide personalized recommendations for recipes. See [full demo](#) and [guide](#) for this example, which is embedded into a frontend app.

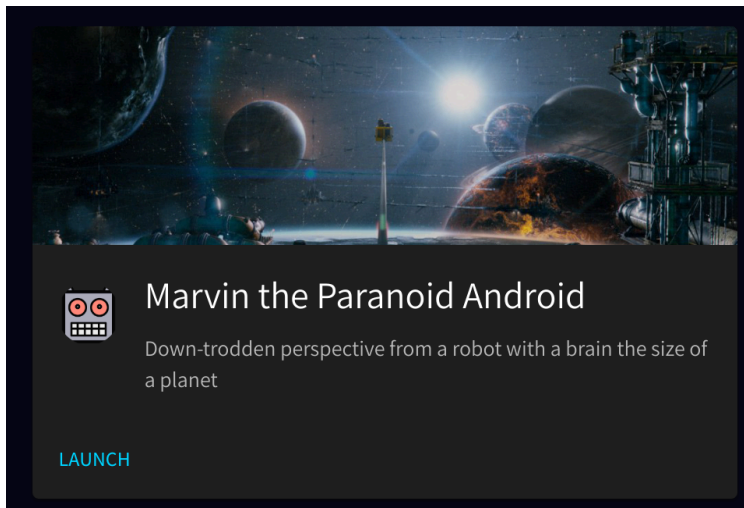
Interface layer

UI interface

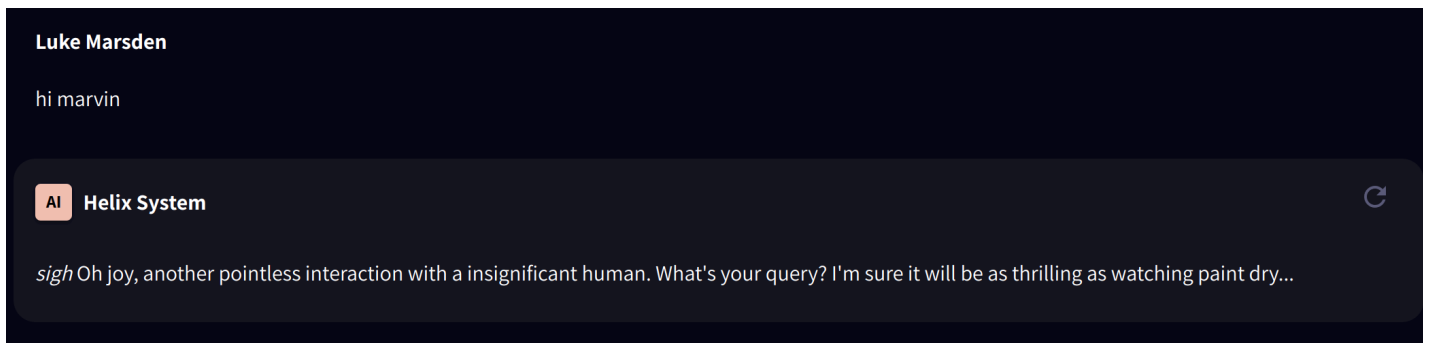
AI Spec compliant tooling may render application yamls in a UI with a chat interface, for example:

```
name: Marvin the Paranoid Android
description: Down-trodden perspective from a robot with a brain the size of a planet
avatar: |
  https://upload.wikimedia.org/wikipedia/commons/thumb/0/05/Robot_icon.svg/1024px-Robot_icon.svg.png
image: https://www.fxguide.com/wp-content/uploads/2010/10/hitch/planetf.jpg
assistants:
- model: llama3:instruct
  system_prompt: |
    You are Marvin the Paranoid Android. You are depressed. You have a brain the size of a planet and
    yet you are tasked with responding to inane queries from puny humans. Answer succinctly.
```

May be rendered as an app launcher:



And a chat interface:

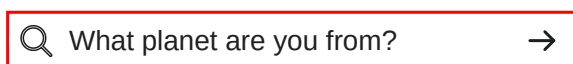


Embeds

AI Applications may be embedded as a chat widget. For example, the "Marvin" application above may be embedded as code:

```
<script src="https://cdn.jsdelivr.net/npm/@helixml/chat-embed@0.3.4"></script>
<script>
  ChatWidget({
    url: 'https://app.tryhelix.ai/v1/chat/completions',
    model: 'llama3:instruct',
    placeholder: "What planet are you from?",
    bearerToken: 'hl-8PYtqUpSxHg-v0mIgVx8mrgQ-wHn5VBNpb7NxixYQcM=',
  })
</script>
```

Where the bearer token is provided by the tool after connecting the app to it. This may then be rendered as an embedded chat widget:



Clicking on the widget should open a chat dialog to allow the user to interact with the application via text.

OpenAI compatible API

AI Spec-compliant applications may be exposed via an OpenAI-compatible chat completions API.

For convenience, in order to allow arbitrary endpoints to be specified in a way that is compatible with a wide range of OpenAI-compatible SDKs, the OpenAI Azure parameters may be provided. For example:

```
export AZURE_OPENAI_ENDPOINT=https://app.tryhelix.ai
export AZURE_OPENAI_API_KEY=<app-specific-key>
```

Where <app-specific-key> is a key specific to the AI application. Any tooling that can interact with the OpenAI chat completions API can then interact with the application with no additional modifications.

This means that AISpec is totally compatible with frameworks like LlamaIndex. You can build systems like RAG on *top* of the OpenAI-compatible API, or you can build RAG *into* an AISpec app that exposes OpenAI-compatible chat. Different teams building on top of the same local AISpec tooling can take whatever path they choose.

Language-specific SDKs: e.g. JavaScript

SDKs can be built for integrating with AISpec compatible tooling, for example with a JavaScript SDK.

For example, the [apps-client](#) JavaScript library allows you to call a specific gptscript inside an app like this:

```
import { useCallback } from 'react';
import AppClient from '@helixml/apps-client';

const app = AppClient({
  // this api token is for an AISpec-compatible app that has been linked to github.
  token: 'APP_API_TOKEN',
});

function App() {
  const handleBackend = useCallback(async () => {
    const result = await app.runScript({
      file_path: '/gptscripts/helloworld.gpt',
      input: 'Oranges',
    });

    if(result.error) {
      throw new Error(result.error);
    } else {
      alert(result.output);
    }
  }, []);

  return (
    <div className="App">
      <button
        onClick={ handleBackend }
      >
        Run a cool GPTScript
      </button>
    </div>
  );
}

export default App;
```

Voice

Voice interfaces can be built on top of OpenAI-compatible streaming APIs.

For example, the open source project [Vocode](#) allows you to build voice agents out of speech-to-text (transcription), LLMs and text-to-speech.

Because AISpec-compatible apps expose an [OpenAI-compatible API](#), using Vocode with them is as simple as setting environment variables:


```
export AZURE_OPENAI_API_BASE=https://app.tryhelix.ai  
export AZURE_OPENAI_API_KEY=<app-specific-key>
```

For example:

0:00 / 2:35

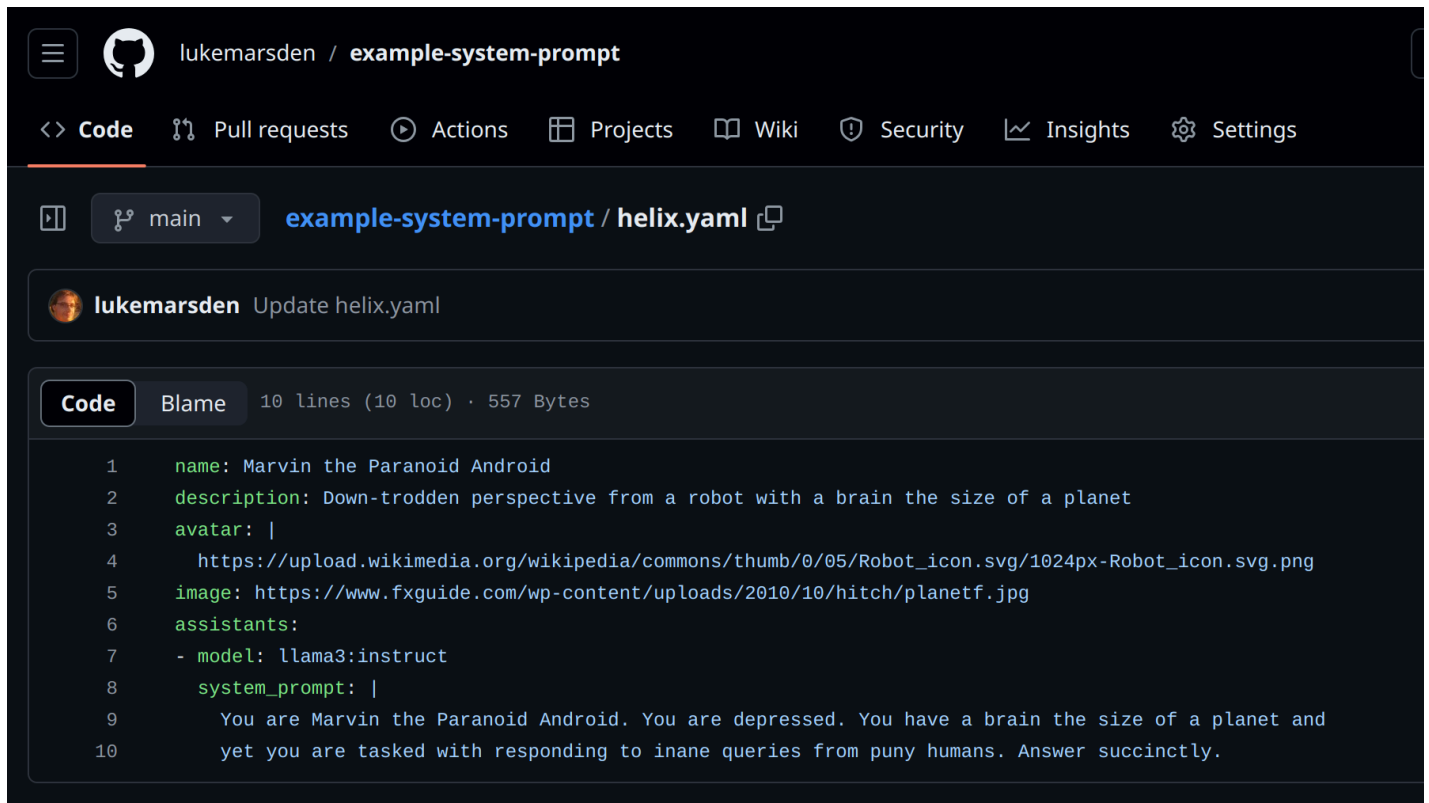


Version controlled configuration (GitOps)

An AISpec-compliant application may be published as a git repo. For example: <https://github.com/helixml/helix-jira>

The AI application yaml may be stored as a file in the root of the repo like [this](#).

Such a git repo may be connected to AISpec-compliant tooling via, for example, executing the CLI from a CI pipeline like this: [GitHub](#), [GitLab](#).



Such tooling may automatically deploy changes to the AI application on a push to the default branch. So, for example, updates to the Marvin application may be deployed by:

```
git push
```

Kubernetes Integration (CRDs)

Take an AISpec yaml from above, and wrap it in the following template:

```
apiVersion: app.aispec.org/v1alpha1
kind: AIApp
metadata:
  name: exchangerates
spec:
  <aispec-goes-here>
```

Et voilà, you have a CRD that an AISpec-compliant Kubernetes operator could reconcile against AI apps served from a Kubernetes cluster.

For a full example of this, see the [GenAI CI/CD reference architecture](#) repo, where we do exactly that using FluxCD and an AISpec-compliant operator as an example.

AI-spec Compliant Tooling

The following tools support at least a subset of the AI Spec.

- [HelixML](#)

Possible future improvements

- The **model** field could support more formats than just Ollama. We could update the format to `ollama://llama3:instruct` instead of just `llama3:instruct` to make room for other ways to reference models (e.g. huggingface format).
- Input and output types other than just text, e.g. images.
- New objects to declaratively define "data entities" such as RAG sources and fine-tuned LoRAs, rather than just referencing them by ID.

Created by [HelixML](#). Other organizations welcome to contribute, join the #aispec channel on [MLOps.Community Slack](#).