NIST AI Policy Builder - Local Development Setup Guide

1. Introduction

This document provides comprehensive, step-by-step instructions for setting up the NIST AI Policy Builder application for local development. The system is composed of two primary services:

1. **Backend Service:** A Python-based FastAPI application responsible for all core logic, including AI model interactions, policy generation, and data management.
2. **Frontend Service:** A Next.js application that provides the user-facing chat interface.

To run the application successfully, both services must be configured and running concurrently. This guide is divided into two parts, one for each service.

2. Part 1: Backend Service Setup (FastAPI)

The backend service orchestrates the application's intelligence, connecting to AI models and managing the conversation flow.

2.1. Prerequisites

Before proceeding, ensure your development environment meets the following requirements:

- Python 3.11 or a later version.
- `pip` , the Python package installer.

2.2. Setup Instructions

Step 1: Navigate to the Project Directory
Open your terminal or command prompt and navigate to the root directory of the backend source code.

Step 2: Create and Activate a Python Virtual Environment
It is a standard best practice to use a virtual environment to isolate project dependencies and avoid conflicts with other Python projects.

- **Create the virtual environment:**

```
conda create --name <env_name>
```

- **Activate the virtual environment:**

  ```
  conda activate <env_name>
  ```

  Your terminal prompt should now indicate that you are operating within the `(conda)` environment.

Step 3: Install Required Dependencies

Install all necessary Python packages as defined in the `requirements.txt` file.

```
pip install -r requirements.txt
```

Step 4: Configure Environment Variables

Secure credentials and environment-specific settings are managed through a `.env` file. This step is critical for connecting to the required AI services.

1. Create a `.env` file by making a copy of the provided example file:

   ```
   cp .env.example .env
   ```

2. Open the newly created `.env` file in a text editor and populate the variables according to the LLM provider you intend to use. The selection between Groq and AWS Bedrock is configured in the `config.yaml` file.

   A) LLM Provider Configuration

   - **Option 1: Using Groq**

     If `config.yaml` is set to use Groq models, populate the following:

     - `GROQ_API_KEY` : Your unique API key from the Groq Console.
     - The `AWS_` variables can be left empty or removed.

   - **Option 2: Using AWS Bedrock**

     If `config.yaml` is set to use Bedrock models, populate the following:

     - `AWS_ACCESS_KEY_ID` : Your AWS IAM user access key ID.
     - `AWS_SECRET_ACCESS_KEY` : Your AWS IAM user secret access key.
     - `AWS_REGION_NAME` : The AWS region where Bedrock is enabled (e.g., `us-east-1`).
     - The `GROQ_API_KEY` variable can be left empty or removed.

   B) Frontend URL Configuration

   This variable is essential for enabling Cross-Origin Resource Sharing (CORS), which allows the frontend application to securely communicate with this backend server.

   - **For Local Development:** Set the URL to the default address of the local frontend server.

```
FRONTEND_URL=http://localhost:3000
```

- **For a Deployed Frontend:** If the frontend is hosted on a server, replace the local address with its public URL (e.g., `FRONTEND_URL=https://your-frontend-domain.com`).

Step 5: Run the Backend Server

With the environment configured, you can now start the FastAPI server using Uvicorn.

```
uvicorn api:app --host 0.0.0.0 --port 8088 --reload
```

- The server will be accessible at `http://localhost:8088`.
- The `--reload` flag enables hot-reloading, automatically restarting the server when code changes are detected, which is highly beneficial for development.

The backend service is now running and ready to handle requests.

3. Part 2: Frontend Service Setup (Next.js)

The frontend service provides a responsive and interactive user interface for the NIST AI Policy Builder.

3.1. Prerequisites

Before proceeding, ensure your development environment meets the following requirements:

-
- Node.js 18.x or a later version.
  `npm`, the Node Package Manager (included with Node.js).

3.2. Setup Instructions

Step 1: Navigate to the Project Directory

Open a new terminal or command prompt window and navigate to the root directory of the frontend source code.

Step 2: Install Required Dependencies

Install all necessary Node.js packages as defined in the `package.json` file.

```
npm install
```

Step 3: Configure Environment Variables

The frontend application must know the URL of the backend API to send requests.

1. Create a `.env` file by making a copy of the provided example file:

   ```
   cp .env.example .env
   ```

2. Open the newly created `.env` file in a text editor and configure the `BACKEND_URL` variable.
   - **For Local Development:** Point this URL to the running backend service.

     ```
     BACKEND_URL=http://localhost:8088
     ```

   - **For a Deployed Backend:** If the backend service is hosted on a server, replace the local address with its public API URL (e.g., `BACKEND_URL=https://api.your-backend-domain.com`).

Step 4: Run the Frontend Development Server

Start the Next.js development server.

```
npm run dev
```

The frontend application will now be running and accessible.

4. Accessing the Application

With both the backend and frontend services running, you can now use the application.

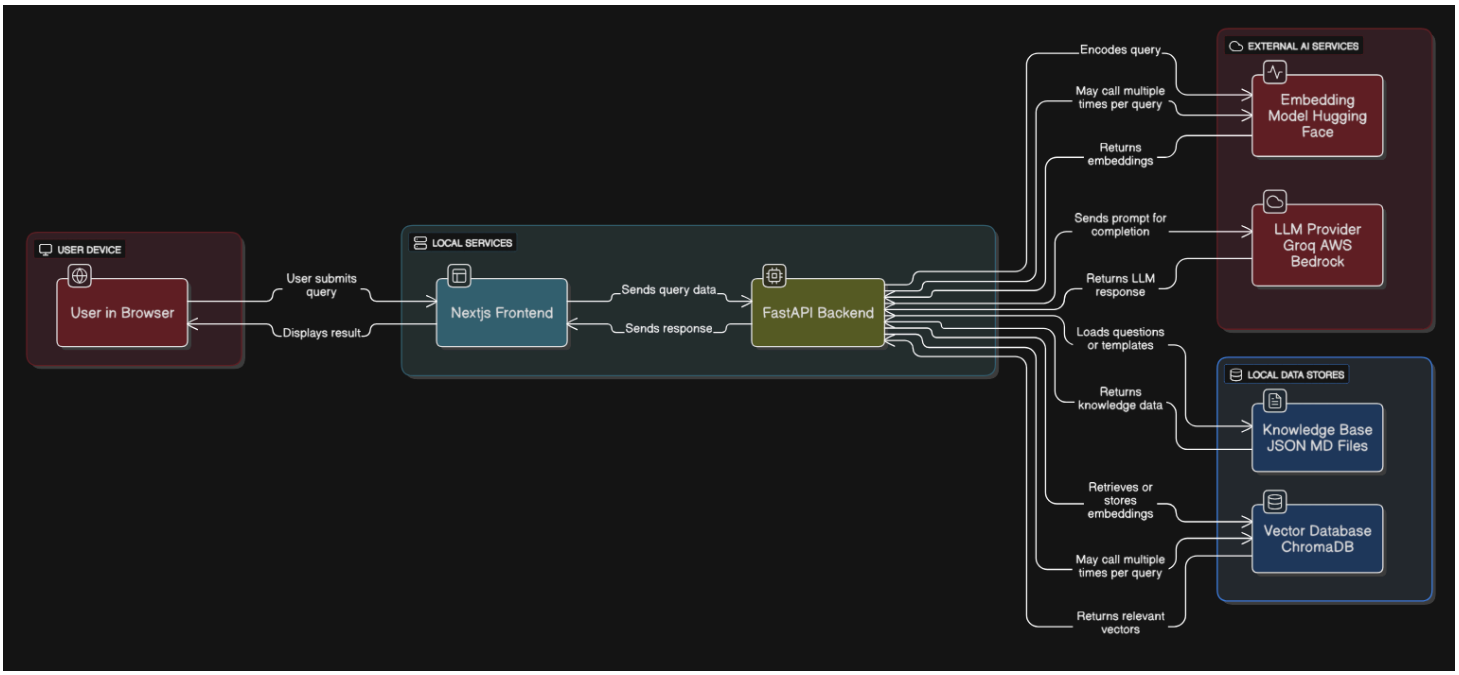
1. Open your preferred web browser.
2. Navigate to the following address: **http://localhost:3000**

You should now see the AI Policy Assistant's chat interface. All interactions within the interface will be processed by your local backend service.

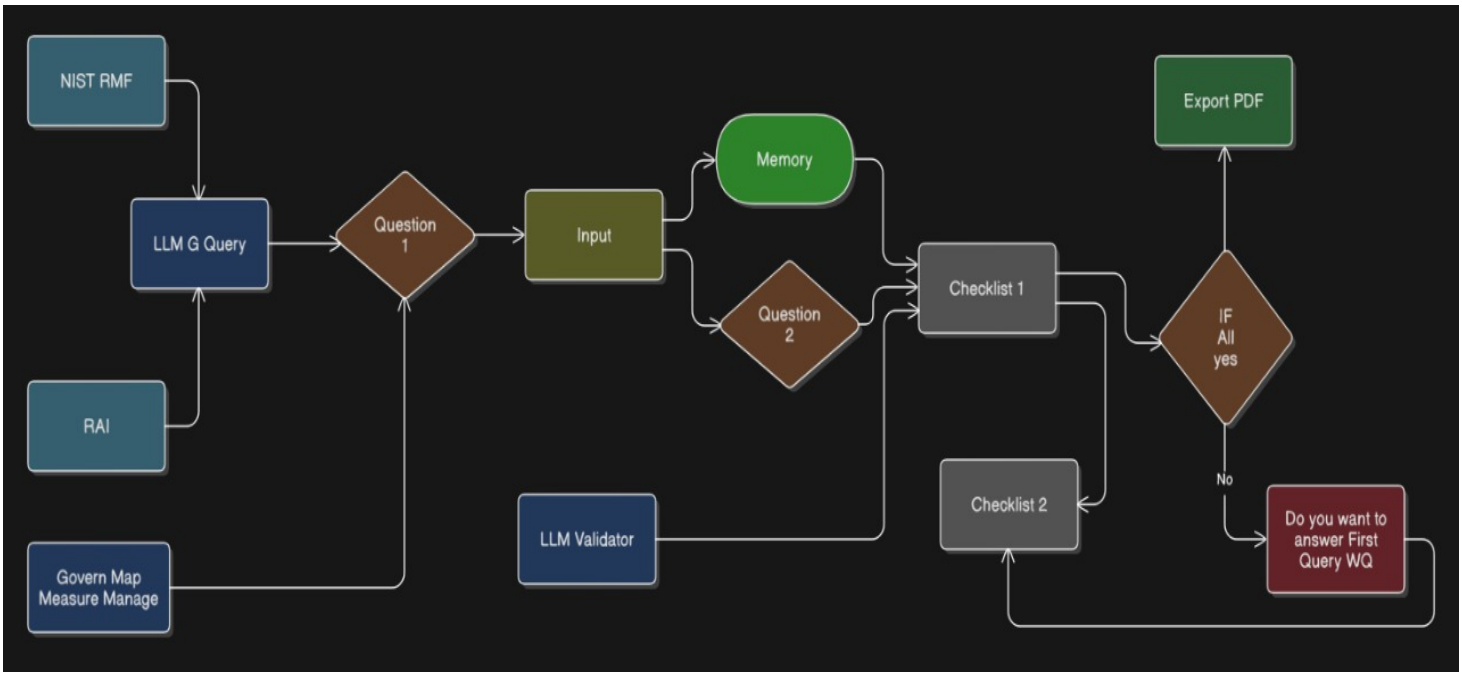
Architecture and Overview

1. Architecture Diagram

1.1 User Flow Diagram



1.2 System Architecture



2. Overview

The application architecture is designed for modularity and maintainability, separating static configuration, application logic, and sensitive credentials. This separation is achieved through two primary files:

1. **`config.yaml`** : The central file for defining the application's behavior, including which AI models to use, where to find data files, and the system prompts that guide the AI agents.
2. **`.env`** : A file for storing sensitive credentials and environment-specific variables, such as API keys. This file is intentionally kept out of version control to protect secret information.

This section details the structure and purpose of these configuration files, as well as the necessary setup for cloud deployment.

3. Core Configuration (`config.yaml`)

The `config.yaml` file is the primary control panel for the application's operational parameters. It is organized into several key sections:

3.1. Model Configuration (`models`)

This section defines the specific AI models used for different tasks within the application.

- `GENERIC_MODEL` : Used for general-purpose questions in the RAG (Retrieval-Augmented Generation) system.
- `QUERY_AGENT_MODEL` : A smaller, faster model used for internal tasks like analyzing user input for clarity.
- `POLICY_GENERATOR_MODEL` : A powerful model optimized for generating the final, structured policy document.
- `VALIDATOR_AGENT_MODEL` : The model responsible for evaluating user answers against predefined criteria.
- `embedding_model_name` : Specifies the sentence-transformer model used to generate vector embeddings for the RAG system.

The system utilizes **LiteLLM** to interface with various model providers. This allows for great flexibility. To switch from AWS Bedrock to another provider (e.g., Groq, OpenAI), you only need to modify the model string to include the new provider's prefix.

- **Example (Bedrock):** `bedrock/meta.llama3-70b-instruct-v1:0`
- **Example (Groq):** `groq/llama3-70b-8192`

For a complete list of supported providers and model identifiers, please refer to the official LiteLLM Documentation.

3.2. Data and Template Paths (`paths`)

This section specifies the file paths for essential data and template files, all of which are located in the `./documents/` directory of the project.

- `policy_json_file` : The path to the JSON file containing the structured set of questions, validators, and example answers that drive the policy-building workflow.
- `generic_json_file` : The path to the JSON file containing general information about the NIST AI RMF. This content is used to power the generic Q&A mode, allowing users to ask informational questions.
- `template_file` : The path to the Markdown (`.md`) file that serves as the master template for generating the final policy document. The AI integrates user answers into this structure.

3.3. System Prompts (`system_prompts`)

This section contains the detailed instructional prompts that define the "persona," rules, and constraints for each AI agent. These prompts are critical for ensuring the AI's responses are accurate, on-topic, and aligned with its designated role.

- `GENERIC_SYSTEM_PROMPT` : Guides the AI in the general Q&A mode.
- `POLICY_SYSTEM_PROMPT` : Instructs the AI during the policy-building workflow.
- `VALIDATOR_SYSTEM_PROMPT` : Defines the strict rules for the answer validation agent, ensuring it returns a structured JSON response.

3.4. Vector Store Collections (`collections`)

This section defines the names for the collections within the ChromaDB vector database. A vector database stores the numerical representations (embeddings) of the text data, enabling efficient similarity searches for the RAG system.

- `generic_collection_name` : The collection for the general NIST AI RMF information.
- `policy_collection_name` : The collection for the policy-related questions and validators.

4. Environment Variables (`.env` file)

While `config.yaml` manages the application's behavior, the `.env` file is used exclusively for storing sensitive credentials and environment-specific URLs. This file is ignored by version control systems (e.g., Git) to prevent accidental exposure of secrets.

As detailed in the setup instructions, this file must be created from `.env.example` and populated with the necessary API keys and the `FRONTEND_URL` .

5. Cloud Deployment: GitHub Actions Secrets

This configuration is only required for automated cloud deployments using the provided CI/CD workflows.

For the GitHub Actions workflow (`manual-ecr-deploy.yml`) to successfully build a Docker image and push it to Amazon Elastic Container Registry (ECR), it must be granted secure access to your AWS account. This is achieved by storing credentials as encrypted secrets within your GitHub repository.

Navigate to your repository on GitHub and go to: **Settings > Secrets and variables > Actions**. Create the following repository secrets:

- **`AWS_ACCESS_KEY_ID`** : Your AWS IAM user's access key ID. This user must have programmatic access and permissions to interact with ECR.
- **`AWS_SECRET_ACCESS_KEY`** : The corresponding secret access key for the IAM user.
- **`AWS_ACCOUNT_ID`** : Your 12-digit AWS account number.
- **`AWS_REGION`** : The AWS region where your ECR repository is located (e.g., `us-east-1`).
- **`ECR_REPOSITORY_NAME`** : The exact name of the ECR repository where the Docker image will be stored.