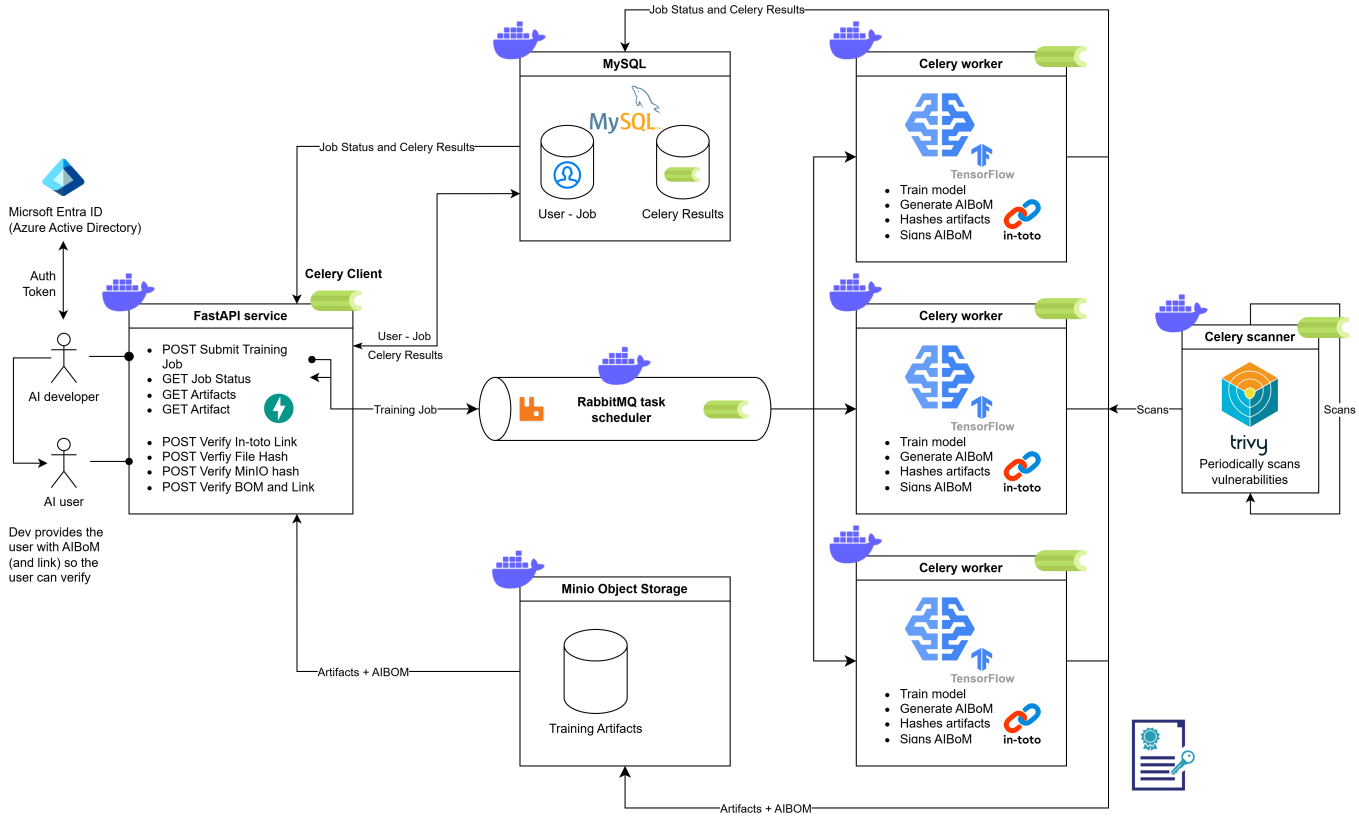
AIBoMGen Platform: Distributed AI Training with AIBoM Generation

Table of Contents

Overview

AIBoMGen Platform is the backend system for distributed AI training with a focus on **trustability**, integrity, reproducibility, and artifact management. It ensures that the training process is fully auditable and tamper-proof by generating an **AI Bill of Materials (AIBoM)**, which includes cryptographic attestations of the training environment, inputs, and outputs.



)

Key features:

- **Automated AIBoM Generation**: Automatically creates an AI Bill of Materials (AIBoM) that documents the training process, including metadata, input/output details, and environment configurations.
- **Supply Chain Integrity**: Leverages in-toto `.link` files to ensure the integrity of training inputs, outputs, and processes.
- **Scalable Distributed Training**: Supports distributed training using Celery workers, enabling efficient execution across multiple nodes.
- **Secure and Isolated Execution**: Runs training jobs in containerized environments to ensure security and prevent interference between jobs.
- **Integrated Artifact Management**: Uses MinIO for efficient storage and retrieval of datasets, models, logs, and other artifacts.
- **Comprehensive Verification**: Provides tools to validate AIBoMs, in-toto `.link` files, and associated artifacts for trust and authenticity.
- **Vulnerability Mitigation**: Regularly scans the system and worker images for vulnerabilities using Trivy to maintain a secure platform.
- **User-Friendly API**: Offers a secure and intuitive API for submitting training jobs and managing artifacts.
- **Azure OAuth Integration**: Implements Azure OAuth for secure API calls, ensuring that only authenticated and authorized users can interact with the system.
- **Extensibility**: Designed to easily update with additional tools, frameworks, and workflows, making it adaptable to various AI training pipelines. Currently focused on basic models using TensorFlow.

Results and Experiments

For results and experiments related to this platform, refer to the **AIBoMGen Experiments repository**.

User Flows

User Flow for an AI Developer (Model Training on the Trusted System)

1. **Prepare Input Files**:
    - The AI developer prepares the necessary input files:
        - **Model file**: Pre-trained model or base model.
        - **Dataset**: Training dataset (e.g., CSV, images, or TFRecord).
        - **Dataset definition**: Metadata describing the dataset (e.g., column names, label information, preprocessing steps).
2. **Submit Training Job**:
    - The developer submits the training job to the trusted system via an API (or UI), providing:
        - Input files (uploaded to MinIO or similar storage).
        - Training parameters (e.g., epochs, batch size, validation split).
        - Optional metadata (e.g., model name, version, description).
3. **Training Execution**:
    - The trusted system:
        - Downloads the input files.
        - Validates the dataset and model compatibility.
        - Trains the model using the provided parameters.
        - Logs the training process and generates metrics.
4. **Generate AIBoM**:
    - After training, the system:
        - Computes hashes for all input and output files (also logs and other intermediary files).
        - Generates the AIBoM (AI Bill of Materials) with optional metadata, model information, input/output artifacts (training data), and environment details (worker/job data).
        - Signs the AIBoM using the system's private key (The platform acts as a trusted entity).
5. **Upload Artifacts**:
    - The system uploads the following artifacts to MinIO or similar storage:
        - Trained model.
        - Training metrics.
        - Logs.
        - Signed In-toto `.link` file with hashes of artifacts.
        - Signed CycloneDX BOM `.json` with reference to In-toto `.link` file.
6. **Share AIBoM**:
    - The developer shares the AIBoM (and `.link` file) with the AI users of their model (along with the public key for verification).

User Flow for an AI User (Verifying the AIBoM)

1. **Receive AIBoM and Artifacts**:
    - The AI user receives:
        - The AIBoM (JSON file).
        - The public key of the trusted system.
2. **Verify the AIBoM**:
    - The user verifies the AIBoM by:
        - Decrypting the signature using the trusted system's public key.
        - Ensuring that the decrypted signature matches the AIBoM content.
    - If the signature verification succeeds, it confirms that:
        - The AIBoM was generated by the trusted system.
        - The AIBoM has not been tampered with.
3. **Verify the Associated in-toto Link File**:
    - The user verifies the `.link` file referenced in the AIBoM by:
        - Using the `/verify_in-toto_link` endpoint to validate the `.link` file against the signed layout.
        - Ensuring that all recorded materials and products match the metadata in the `.link` file.
4. **Verify Artifacts Against the AIBoM**:
    - The user can verify the integrity of artifacts (e.g., datasets, models) by:
        - Using the `/verify_file_hash` endpoint to compare the hash of an uploaded file against the recorded metadata in the `.link` file.
        - Using the `/verify_minio_artifacts` endpoint to validate that materials and products stored in MinIO match the metadata in the `.link` file.
5. **Verify the CycloneDX AIBoM**:
    - The user verifies the authenticity and integrity of the CycloneDX AIBoM by:
        - Using the `/verify_bom_and_link` endpoint to validate the AIBoM's cryptographic signature using the platform's public key.
        - Ensuring that the AIBoM's metadata and associated `.link` file are consistent and untampered.
6. **Trust Decision**:
    - If all verification steps succeed (or just step 5), the user can trust the AI system and proceed to use the trained model.
    - If any verification step fails, the user should reject the model and notify the developer.

Requirements

Prerequisites (You need to install)

- **Docker**: For containerized deployment.
- **Docker Compose**: To orchestrate the containers.

- **NVIDIA Container Toolkit**: To allow worker containers to use the host machine's GPU. Install it by following the official NVIDIA documentation. For windows make sure you are running docker on WSL2.
- **Environment Variables**: Create a `.env` file in the aibomgen-platform directory to configure the system (see **Setup** for details).
- **Platform Secrets**: Generate platform secrets using the `generate_in-toto_signed_layout.py` script provided in the repository. This script generates a private-public key pair and a layout for supply chain verification and signing, ensuring the integrity and trustability of the system's supply chain.

Requirements in Docker containers (No need to install manually)

API Container

- **Python 3.9**: For FastAPI and Celery integration.
- **FastAPI**: For building the API.
- **Uvicorn**: For running the FastAPI application.
- **celery[sqlalchemy]**: For task scheduling and distributed execution.
- **Pydantic**: For data validation.
- **Pydantic-Settings**: For managing application settings.
- **Python-Multipart**: For handling file uploads.
- **Python-Dotenv**: For loading environment variables.
- **Boto3**: For interacting with AWS S3-compatible storage (e.g., MinIO).
- **PyYAML**: For parsing YAML files.
- **SlowAPI**: For rate-limiting API requests.
- **FastAPI-Azure-Auth**: For integrating Azure AD authentication.
- **SQLAlchemy**: For database ORM functionality.
- **MySQL-Connector-Python**: For connecting to MySQL databases.
- **In-toto**: For supply chain security and metadata verification.
- **Cryptography**: For signing and verifying AIBoMs.
- **CycloneDX-Python-Lib[Validation]**: For generating and validating CycloneDX SBOMs.

Worker Container

- **Celery[sqlalchemy]**: For task scheduling and distributed execution.
- **Flower**: For monitoring Celery workers.
- **python-dotenv**: For loading environment variables.
- **PyYAML**: For parsing dataset definitions.
- **Pandas**: For dataset preprocessing.
- **Cryptography**: For signing the AIBoM.
- **cyclonedx-python-lib[validation]**: For generating and validating CycloneDX SBOMs.
- **Boto3**: For interacting with AWS S3-compatible storage (e.g., MinIO).
- **cffi**: For interacting with C libraries.
- **in-toto**: For supply chain security and metadata verification.
- **psutil**: For system monitoring and resource management.
- **nvidia-ml-py3**: For monitoring NVIDIA GPU usage.
- **Docker**: For containerized execution of training jobs.
- **mysql-connector-python**: For connecting Celery with the mysql results backend.
- **numpy**: For efficient array operations.

Scanner Container

- **Celery[sqlalchemy]**: For task scheduling and distributed execution.
- **python-dotenv**: For loading environment variables.
- **Docker**: For containerized execution of scanning tasks.
- **Boto3**: For interacting with AWS S3-compatible storage (e.g., MinIO).
- **mysql-connector-python**: For connecting Celery with the mysql results backend.

Repository Structure

```
aibomgen-platform/
├── .env                                # Environment variables configuration
├── docker-compose.yml                  # Docker Compose configuration
├── README.md                           # This documentation
├── README.pdf                          # PDF version of the documentation
│
├── api/                                # API service
│   ├── app.py                          # FastAPI application
│   ├── auth_utils.py                   # Authentication utilities
│   ├── celery_config.py                # Celery configuration
│   ├── celery_utils_endpoints.py       # Celery utility endpoints
│   ├── database.py                     # Database connection and models
│   ├── developer_endpoints.py          # Endpoints for AI developers
│   ├── Dockerfile                      # API service Docker configuration
│   ├── models.py                       # ORM models
│   ├── requirements.txt                # API dependencies
│   └── verifier_endpoints.py           # Endpoints for AI users (verification)
├── flower/                             # Flower monitoring service
│   ├── celery_config.py                # Celery configuration for Flower
│   └── Dockerfile                      # Flower service Docker configuration
├── mysql/                              # MySQL database initialization
│   └── init-multiple-dbs.sql           # SQL script for initializing both
databases
├── scanner/                            # Scanner service
│   ├── celery_config.py                # Celery configuration for Scanner
│   ├── Dockerfile                      # Scanner service Docker configuration
│   ├── requirements.txt                # Scanner dependencies
│   └── tasks.py                        # Celery tasks for scanning
├── secrets/                            # Platform secrets
│   ├── layout.json                     # In-toto layout file
│   ├── signed_layout.json              # Signed In-toto layout
│   ├── worker_private_key.pem          # Worker private key
│   └── worker_public_key.json          # Worker public key
├── shared/                             # Shared utilities
│   ├── in_toto_utils.py                # In-toto helper functions
│   ├── minio_utils.py                  # MinIO helper functions
│   └── zip_utils.py                    # ZIP file validation and extraction
utilities
├── utils/                              # Utility scripts
│   ├── generate_cifar_test_files.py    # Script to generate CIFAR test files
│   ├── generate_in-toto_signed_layout.py  # Script to generate signed In-toto
```

```
layout
│   ├── generate_mnist_test_files.py       # Script to generate MNIST test files
│   ├── generate_private_key_deprecated.py # Deprecated script for generating
private keys
│   ├── generate_tabular_test_files.py     # Script to generate tabular test files
│   ├── generate_tfrecord_test_files.py    # Script to generate TFRecord test
files
│   └── open-dashboards.py                 # Script to open monitoring dashboards
│   └── old_test_files/                    # Deprecated test file scripts
│       ├── deprecated_model_filemaker.py
│       ├── test_files.py
│       ├── test_files2.py
│       └── test_files_img.py
└── worker/                                # Worker service
    ├── bom_data_generator.py              # BOM data generation logic
    ├── celery_config.py                   # Celery configuration for Worker
    ├── Dockerfile                         # Worker service Docker configuration
    ├── entrypoint.sh                      # Worker entrypoint script
    ├── environment_extractor.py           # Extracts environment details for
AIBoM
    ├── in_toto_link_generator.py          # Generates In-toto link files
    ├── requirements.txt                   # Worker dependencies
    ├── tasks.py                           # Celery tasks for training
    ├── training_logic.py                  # Training logic implementation
    └── transform_to_cyclonedx.py          # CycloneDX BOM transformation logic
```

Setup

1. Clone the Repository

```
git clone <repository-url>
cd AIBoMGen
cd aibomgen-platform
```

2. Start Docker Daemon

- **Linux**:

  ```
  sudo systemctl start docker
  ```

- **Windows**: Start Docker Desktop.

3. Configure Environment Variables

Create a `.env` file in the root directory with the following variables:

```
RABBITMQ_USER=rmq_user
RABBITMQ_PASSWORD=rmq_password
FLOWER_BASIC_AUTH=admin:admin
CELERY_BROKER_URL=amqp://rmq_user:rmq_password@rabbitmq:5672//
CELERY_RESULT_BACKEND=db+mysql+mysqlconnector://aibomgen_user:aibomgen_password@my
sql:3306/celery_results
WORKER_IMAGE_NAME=aibomgen-platform-worker:latest
MINIO_ROOT_USER=minio_user
MINIO_ROOT_PASSWORD=minio_password
MINIO_ENDPOINT=http://minio:9000
TRAINING_BUCKET=training-jobs
WORKER_SCANS_BUCKET=worker-scans
SCANNER_SCANS_BUCKET=scanner-scans
ENABLE_SCANNER=false
AUTH_ENABLED=true
APP_CLIENT_ID=<your-app-client-id>
OPENAPI_CLIENT_ID=<your-openapi-client-id>
MYSQL_ROOT_PASSWORD=root_password
MYSQL_DATABASE=aibomgen
MYSQL_USER=aibomgen_user
MYSQL_PASSWORD=aibomgen_password
DATABASE_URL=mysql+mysqlconnector://aibomgen_user:aibomgen_password@mysql:3306/aib
omgen
REDIS_PASSWORD=redis_password
```

Note: For testing purposes, it is recommended to set `AUTH_ENABLED=false` and `ENABLE_SCANNER=false` to simplify the setup and avoid additional authentication or scanning configurations. To test the frontend authentication HAS to be enabled! How to do that is explained in OAuth Setup.

4. Generate Platform Secrets for Signing

Run the `generate_in-toto_signed_layout.py` script located in the `utils/` directory. This script will generate a private-public key pair and a signed layout for supply chain verification. The private key will be used to cryptographically sign the AIBoM, ensuring trustability.

5. Start Docker Compose

```
docker-compose up --build
```

Note: Ensure that the `worker/entrypoint.sh` file uses LF (Line Feed) line endings. If the file has CRLF (Carriage Return + Line Feed) line endings, the workers may fail to start. You can configure this in your text editor or use the following command to convert the line endings:

```
sed -i 's/\r$//' worker/entrypoint.sh
```

OAuth Setup

To enable Azure OAuth for secure API calls, follow the instructions provided in the FastAPI Azure Auth documentation. This guide will help you configure your Microsoft Entra ID (Azure Active Directory) application and integrate it with AIBoMGen. For testing a B2C setup was used but ofcourse other setups are also possible.

Ensure the following environment variables are set in your `.env` file:

```
AUTH_ENABLED=true
APP_CLIENT_ID=<your-app-client-id>
OPENAPI_CLIENT_ID=<your-openapi-client-id>
```

For testing purposes, you can disable authentication by setting `AUTH_ENABLED=false`.

Note: for using the frontend authentication HAS to be enabled! Additional Microsoft Entra ID setup is required! Refer to the frontend setup README.

If there are problems setting up the authentication with the frontend or backend, feel free to contact me: contact

---

Usage

API Endpoints

The FastAPI service runs on http://localhost:8000. Below are the **key endpoints**:

1. Submit a Training Job

- **Endpoint**: `POST developer/submit_job_by_model_and_data`
- **Request Body**:
  - **Files**:
    - `model`: The model file to be trained (e.g., `.keras` for TensorFlow framework).
    - `dataset`: The dataset file for training (e.g., `.csv` or `.zip` for image data).
    - `dataset_definition`: The dataset definition file (e.g., `.yaml`).
  - **Metadata**:
    - `framework`: Currently, only `TensorFlow 2.16.1` is supported.
    - `model_name`: Name of the model (optional).
    - `model_version`: Version of the model (optional).
    - `model_description`: Description of the model (optional).
    - `author`: Author of the model (optional).
    - `model_type`: Type of the model (e.g., Image Classification) (optional).
    - `base_model`: Base model used (e.g., ResNet50) (optional).
    - `base_model_source`: Source URL of the base model (optional).
    - `intended_use`: Intended use of the model (optional).
    - `out_of_scope`: Out-of-scope use cases (optional).
    - `misuse_or_malicious`: Misuse or malicious use cases (optional).
    - `license_name`: License name for the model (optional).

    - **Training Parameters**:
        - `epochs`: Number of epochs to train (default: 50).
        - `validation_split`: Fraction of data to use for validation (default: 0.2).
        - `initial_epoch`: Epoch at which to start training (default: 0).
        - `batch_size`: Size of the batches of data (default: 32).
        - `steps_per_epoch`: Total number of steps per epoch (optional).
        - `validation_steps`: Number of steps for validation (optional).
        - `validation_freq`: Frequency of validation runs (default: 1).

- Testing files can be generated using the provided python scripts in the `utils/` directory.

- **Example dataset_definition.yaml**:

  More examples are available in the AIBoMGen-experiments repo.

```
columns:
   feature1: float
   feature10: float
   feature11: float
   feature12: float
   feature13: float
   feature2: float
   feature3: float
   feature4: float
   feature5: float
   feature6: float
   feature7: float
   feature8: float
   feature9: float
   label: int
input_shape:
- 13
label: label
output_shape:
- 1
preprocessing:
   clip:
   - 0.0
   - 1.0
   normalize: true
   scale: 1.0
```

- **Response**:

```
{
  "job_id": "123e4567-e89b-12d3-a456-426614174000",
  "status": "Training started",
  "unique_dir": "6251847e-710f-47a3-b809-070518e4b1b9"
}
```

2. Check Job Status

- **Endpoint**: `GET developer/job_status/{job_id}`
- **Response**:

```
{
  "job_id": "123e4567-e89b-12d3-a456-426614174000",
  "status": "completed",
  "result": {
    "training_status": "training job completed",
    "unique_dir": "6251847e-710f-47a3-b809-070518e4b1b9",
    "job_id": "123e4567-e89b-12d3-a456-426614174000",
    "message": "Training completed successfully and AIBoM generated.",
    "output_artifacts": [
      "trained_model.keras",
      "metrics.json",
      "logs.txt",
      "aibom.json",
      "aibom.sig"
    ]
  }
}
```

3. Retrieve Job Artifacts

- **Endpoint**: `GET developer/job_artifacts/{job_id}`
- **Response**:

```
{
  "job_id": "123e4567-e89b-12d3-a456-426614174000",
  "artifacts": [
    ".../trained_model.keras",
    ".../etrics.json",
    ".../logs.txt",
    ".../aibom.json",
    ".../aibom.sig"
  ]
}
```

4. Download a Specific Artifact

- **Endpoint**: `GET developer/job_artifacts/{job_id}/{artifact_name}`
- **Response**:

```
{
  "artifact_name": "trained_model.keras",
```

```
  "url": "http://minio:9000/aibomgen/6251847e-710f-47a3-b809-
070518e4b1b9/trained_model.keras?X-Amz-Algorithm=..."
}
```

5. Verify an In-toto Link File

- **Endpoint**: `POST verifier/verify_in-toto_link`
- **Request Body**:
  - **File**:
    - `link_file`: The in-toto link file to verify (e.g., `run_training.<keyid>.link`).
- **Response**:

```
{
  "status": "success",
  "message": "Verification successful.",
  "details": {
    "layout_signature": "Verified",
    "layout_expiration": "Valid",
    "link_signatures": "Verified",
    "threshold_verification": "Met",
    "artifact_rules": "All rules satisfied"
  }
}
```

6. Verify a File Hash Against an In-toto Link File

- **Endpoint**: `POST verifier/verify_file_hash`
- **Request Body**:
  - **Files**:
    - `link_file`: The in-toto link file (e.g., `run_training.<keyid>.link`).
    - `uploaded_file`: The file to verify (e.g., a model or metrics file).
- **Response**:

```
{
  "status": "success",
  "message": "Hash matches the recorded metadata.",
  "details": {
    "file_name": "trained_model.keras",
    "hash": {
      "sha256":
"d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2"
    }
  }
}
```

7. Verify MinIO Artifacts Against an In-toto Link File

- **Endpoint**: `POST verifier/verify_minio_artifacts`
- **Request Body**:
  - **File**:
    - `link_file`: The in-toto link file (e.g., `run_training.<keyid>.link`).
- **Response**:

```
{
  "status": "success",
  "verified_materials": ["2809d3f5-72d8-4097-932c-401f3433c255/model.h5"],
  "verified_products": ["2809d3f5-72d8-4097-932c-
401f3433c255/trained_model.keras"],
  "mismatched_materials": [],
  "mismatched_products": []
}
```

8. Verify a CycloneDX BOM and Associated In-toto Link File

- **Endpoint**: `POST verifier/verify_bom_and_link`
- **Request Body**:
  - **File**:
    - `bom_file`: A signed CycloneDX BOM file (JSON format).
- **Response**:

```
{
  "status": "success",
  "message": "BOM and .link file verification successful.",
  "details": {
    "bom_signature": "Verified",
    "link_file": "Verified"
  }
}
```

AIBoM Generation (CycloneDX format)

The **AI Bill of Materials (AIBoM)** is a JSON document that captures:

- **Environment Details**: Captures details such as OS, Python version, TensorFlow version, CPU count, memory, disk usage, GPU information, Docker container details, and vulnerability scan results.
- **Input Files**: Includes hashes and metadata for datasets and dataset definitions used during training.
- **Output Files**: Includes hashes and metadata for the trained model and metrics generated during training.
- **Configuration**: Captures training parameters (e.g., epochs, batch size) and optional metadata (e.g., model name, version, description).

- **Attestations**: References an in-toto `.link` file for artifact integrity verification.
- Example AIBoM JSON File
- Example In-toto Link File

Trustability Features

1. **Automated AIBoM Generation**:
    - The AIBoM is generated automatically by the system during the training process. Users cannot modify or bypass this step, ensuring a consistent and tamper-proof workflow.
2. **Cryptographic Signing**:
    - The AIBoM is signed using a private key, creating a digital signature that ensures the authenticity and integrity of the document. This guarantees that the AIBoM originates from the trusted system.
3. **Verification**:
    - The generated AIBoM and its signature can be verified using the corresponding public key. This ensures that the training process and its outputs have not been tampered with and remain trustworthy.
    - Additionally, the AIBoM references an **in-toto `.link` file** for artifact integrity verification. The `.link` file contains cryptographic hashes of all input and output artifacts, ensuring that the materials and products used during training match the recorded metadata. Verification of the `.link` file ensures the integrity of the supply chain.
4. **Controlled API**:
    - Users interact with the system exclusively through a secure API. This prevents arbitrary code execution, ensuring that users cannot tamper with the training process, bypass the AIBoM generation, or interfere with the system's infrastructure. All interactions are logged for auditability.
5. **Containerized Workers**:
    - Training jobs are executed in isolated Docker containers. This ensures that each job runs in a secure and controlled environment, preventing users from accessing or modifying other parts of the system. The containerized approach also enhances scalability and resource management.

Dashboards

The backend system provides several dashboards for monitoring and managing the components of AIBoMGen:

1. **API (FastAPI)**:
    - **URL**: http://localhost:8000/docs

- The FastAPI dashboard provides interactive API documentation using Swagger UI. You can test endpoints, view request/response formats, and explore the API.

2. **RabbitMQ**:
   - **URL**: http://localhost:15672
   - RabbitMQ's management UI allows you to monitor message queues, exchanges, and worker activity. Use the credentials defined in your `.env` file (`RABBITMQ_USER` and `RABBITMQ_PASSWORD`) to log in.

3. **Flower (Celery)**:
   - **URL**: http://localhost:5555
   - Flower provides a real-time monitoring dashboard for Celery workers. You can view task statuses, worker activity, and task execution details.

4. **MinIO**:
   - **URL**: http://localhost:9001
   - MinIO's web interface allows you to manage datasets, models, and artifacts stored in the object storage. Use the credentials defined in your `.env` file (`MINIO_ROOT_USER` and `MINIO_ROOT_PASSWORD`) to log in.
