

AlBoMGen: Distributed AI Training with AlBoM Generation

Overview

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

Key features:

- **Dataset Integrity Verification:** Hashing datasets and inputs to ensure consistency and prevent tampering.
 - **AlBoM Generation with Attestations:** Captures metadata, input/output hashes, and environment details, signed with a private key to ensure authenticity and trustability.
 - **Tamper-Proof Workflow:** The AlBoM generation process is automated and cannot be altered by the user, ensuring the integrity of the training pipeline.
 - **Secure API:** The system uses a controlled API to prevent arbitrary code execution, ensuring that users cannot tamper with the training process, shut down containers, or bypass the AlBoM generation.
 - **Containerized Workers:** Training jobs are executed in isolated Docker containers, providing an additional layer of security and ensuring that the system remains stable and safe from user interference.
 - **Distributed Training:** Uses Celery workers for scalable training jobs.
 - **MinIO Integration:** Handles file storage and retrieval for datasets, models, and logs.
-

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.
 - AIBoM (JSON file).
 - AIBoM signature.

6. **Share AIBoM:**

- The developer shares the AIBoM and its signature with the AI users of his 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 AIBoM signature.
 - 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. **Check Metadata:**

- The user inspects the AIBoM metadata to ensure everything looks valid.

4. **Trust Decision:**

- If the AIBoM passes all verification steps, the user can trust the AI system and proceed to use the trained model.
- If the AIBoM fails verification, the user should reject the model and notify the developer.

5. Optionally verify artifacts (datasets, model, etc.) provided to the user

- Recompute the hashes of the provided artifacts (datasets, model, etc.).
 - Compare the recomputed hashes with the hashes recorded in the **verified** AIBoM.
 - If the hashes match, the artifacts given are authentic and untampered.
-

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](#).
- **Environment Variables**: Create a `.env` file in the root directory to configure the system (see **Setup** for details).
- **Private Key**: A valid `private-key.pem` file must be placed in the worker container/directory. This key is used to cryptographically sign the AIBoM as an authority of the platform (e.g., the EU for compliance with the EU AI Act).

Handled by Docker (API and Worker Containers)

API Container

- **Python 3.9**: For FastAPI and Celery integration.
- **FastAPI**: For building the API.
- **Uvicorn**: For running the FastAPI application.
- **Celery**: For task scheduling and distributed execution.
- **Pydantic**: For data validation and settings management.
- **Requests**: For making HTTP requests.
- **Python-Multipart**: For handling file uploads.
- **Python-Dotenv**: For loading environment variables.
- **Boto3**: For interacting with AWS S3-compatible storage (e.g., MinIO).

Worker Container

- **Python 3.9**: For TensorFlow and Celery integration.
- **Celery**: For task scheduling and distributed execution.
- **Flower**: For monitoring Celery workers.
- **Python-Dotenv**: For loading environment variables.
- **TensorFlow**: For model training.
- **PyYAML**: For parsing dataset definitions.
- **Pandas**: For dataset preprocessing.
- **Cryptography**: For signing the AIBoM.

- **CycloneDX-Python-Lib**: For generating CycloneDX SBOMs.
- **Boto3**: For interacting with AWS S3-compatible storage (e.g., MinIO).

Repository Structure

```

AIBoMGen/
├── api/
│   ├── Dockerfile           # API service Docker configuration
│   ├── requirements.txt      # API dependencies
│   ├── app.py               # FastAPI application
│   └── __init__.py          # API package initialization
├── worker/
│   ├── Dockerfile           # Worker service Docker configuration
│   ├── requirements.txt      # Worker dependencies
│   ├── tasks.py             # Celery tasks for training
│   ├── celery_config.py     # Celery configuration
│   ├── transform_to_cyclonedx.py # CycloneDX BOM transformation logic
│   ├── bom_data_generator.py # BOM data generation logic
│   └── entrypoint.sh         # Worker entrypoint script
├── shared/
│   ├── minio_utils.py       # MinIO helper functions
│   └── zip_utils.py         # ZIP file validation and extraction utilities
├── utils/
│   ├── open-dashboards.py    # Script to open monitoring dashboards
│   ├── generate_private_key.py # Script to generate private keys
│   └── generate_test_files/  # Scripts for generating test models and
├── datasets
│   ├── test_files.py
│   ├── test_files2.py
│   ├── test_filesIMG.py     # CNN test model (recommended for testing)
│   └── model_filemaker.py
├── docker-compose.yml       # Docker Compose configuration
├── .env                     # Environment variable configuration
├── .gitignore               # Git ignore rules
├── .gitattributes           # Git attributes configuration
└── README.md                # Project documentation

```

Setup

1. Clone the Repository

```

git clone <repository-url>
cd AIBoMGen

```

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=rpc://
MINIO_ROOT_USER=minio_user
MINIO_ROOT_PASSWORD=minio_password
MINIO_ENDPOINT=http://minio:9000
MINIO_BUCKET_NAME=aibomgen
```

4. Add a Private Key for Signing

Place a valid `private-key.pem` file in the `worker/` directory or ensure it is mounted into the worker container. This key is used to cryptographically sign the AIBoM, ensuring trustability and compliance with regulations (e.g., the EU AI Act). The `utils` directory has a python script `generate_private_key.py` to let you generate one for testing purposes.

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
```

Usage

API Endpoints

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

1. Submit a Training Job

- **Endpoint:** `POST /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`:**

```
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 /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 /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 /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=..."
}
```

UNDER DEVELOPMENT

AIBoM Generation

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

- **Environment Details:** Python version, TensorFlow version, and system metadata.
- **Input Files:** Hashes of datasets and model files used for training.
- **Output Files:** Hashes of generated artifacts (e.g., trained model, metrics).
- **Configuration:** Training parameters such as epochs, batch size, and learning rate.

Example AIBoM

```
{
  "environment": {
    "python_version": "3.9",
    "tensorflow_version": "2.12.0",
    "request_time": "2025-04-16 17:00:00",
    "job_id": "123e4567-e89b-12d3-a456-426614174000"
  },
  "inputs": {
```


[illegible]

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.

2. Cryptographic Signing:

- The AIBoM is signed using a private key, creating a digital signature that ensures the authenticity and integrity of the document.

3. Verification:

- The generated AIBoM and its signature can be verified using the corresponding public key, ensuring that the training process was not tampered with.

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.

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.

Dashboards

The 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.
-