VIA Microservices 2.0 DP User Guide

July 2024

RN # DU-11946-001

Table of Contents

Getting Started

Advances in AI video understanding and interaction have the potential to revolutionize how we access, analyze, and interact with video content in various domains. These AI models are capable of:

- Video captioning-Generating text descriptions or summary of videos.
- Question answering-Answering questions about a video's content.
- Video retrieval-Finding specific videos (highlights) based on text queries.
- Action recognition-Identifying actions happening in the video.

The current release of VIA microservices demonstrates **Video Summarization** with accelerated performance on NVIDIA hardware.

Features

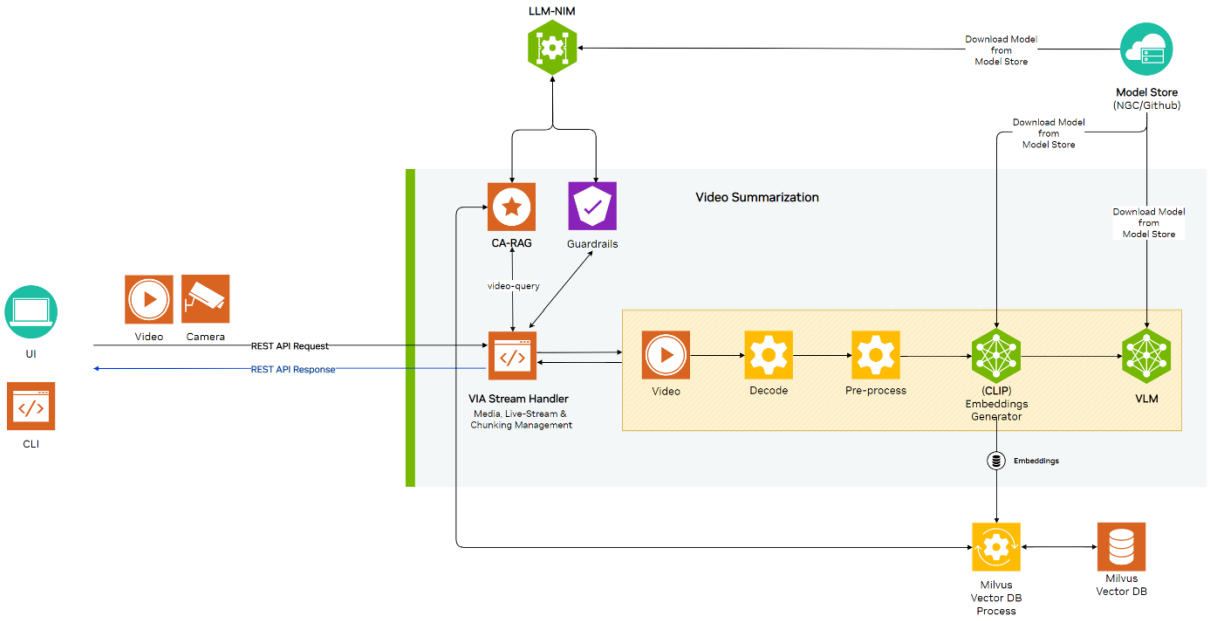
VIA microservice supports video upload, live stream support, summarizing on video files and live streams with various configuration options.

Features:

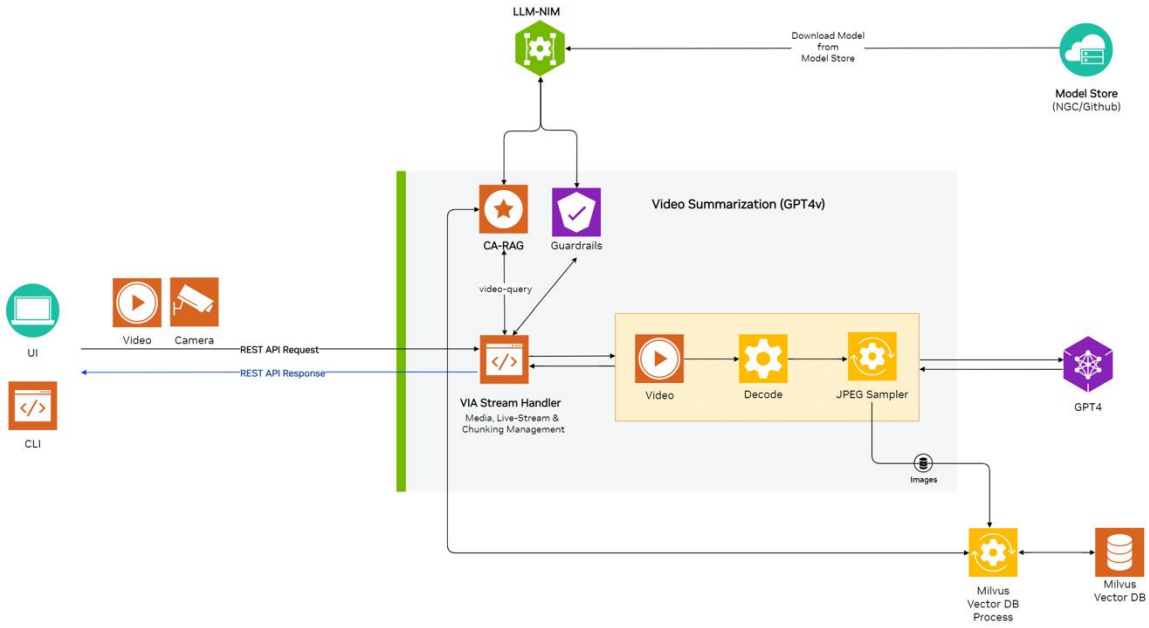
- Faster/Quick Long video processing – create and process multiples video chunks in parallel.
- Live Stream (RTSP) support
- Summarization for videos and live streams
- TRT-LMM support for VITA only
- 1 Node multiple GPU support
- Auto scaling across GPUs
- Context aware RAG support for enhanced accuracy & details
- Support for GPT-4v Turbo and GPT-4o
- Use OpenAI Compatible hosted VLM models
- Drop-in support for custom VLMs
- Guardrails support
- GPU supported – H100, A100, L40, L40S, A6000

VIA Microservice Architecture

VIA Using locally executed VLMs



VIA Using GPT4 (Default configuration)

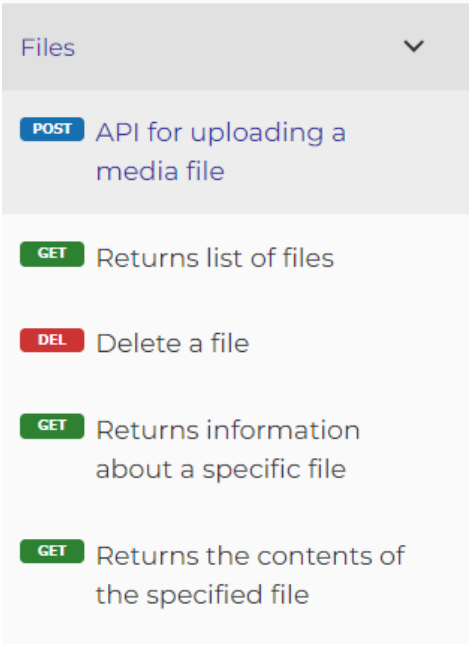


API Documentation

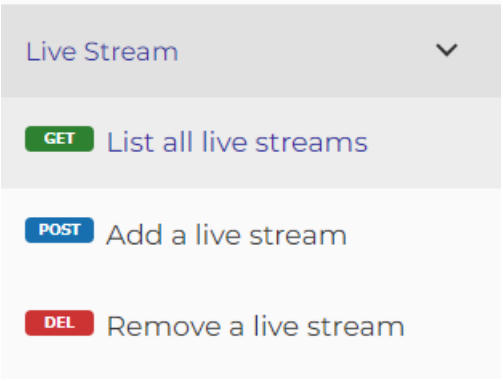
Detailed VIA API documentation is available as OpenAPI spec at https://github.com/NVIDIA-AI-IOT/via-engine/blob/main/api_spec/swagger.json.

Following is a summary of the available APIs.

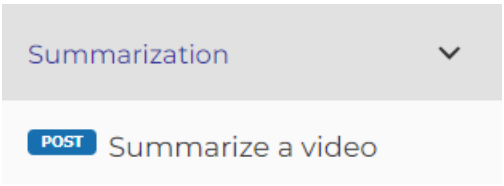
REST APIs for File Management



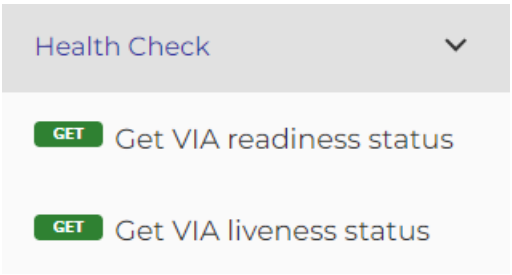
REST API for Live-Stream Management



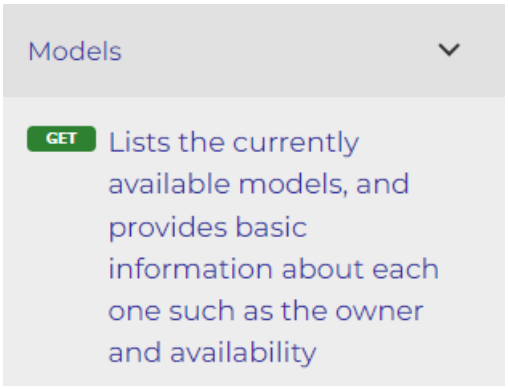
REST API for Video Summarization



REST API for Health Check



REST API for Models



Quick Start Guide

Supported Platforms

Following Nvidia GPUs are supported:

- L40 / L40s
- H100
- A100 80GB / 40GB
- A6000

Prerequisites

- 65+ GB system memory (For locally executed VLMs)
- 40+ GB GPU memory (For locally executed VLMs)
- Ubuntu 22.04
- NVIDIA driver 535.161.08
- Docker Engine
- NVIDIA Container Toolkit
- NVIDIA API Key
- NGC API Key
- OpenAI API Key (Only for GPT4)

Install the NVIDIA Driver

- Download and install using NVIDIA driver 535.161.08 from NVIDIA Unix drivers page at: https://www.nvidia.cn/Download/driverResults.aspx/222416/en-us/
- Run the following commands:

```
chmod 755 NVIDIA-Linux-x86_64-535.161.08.run
sudo ./NVIDIA-Linux-x86_64-535.161.08.run --no-cc-version-check
```

Install Docker

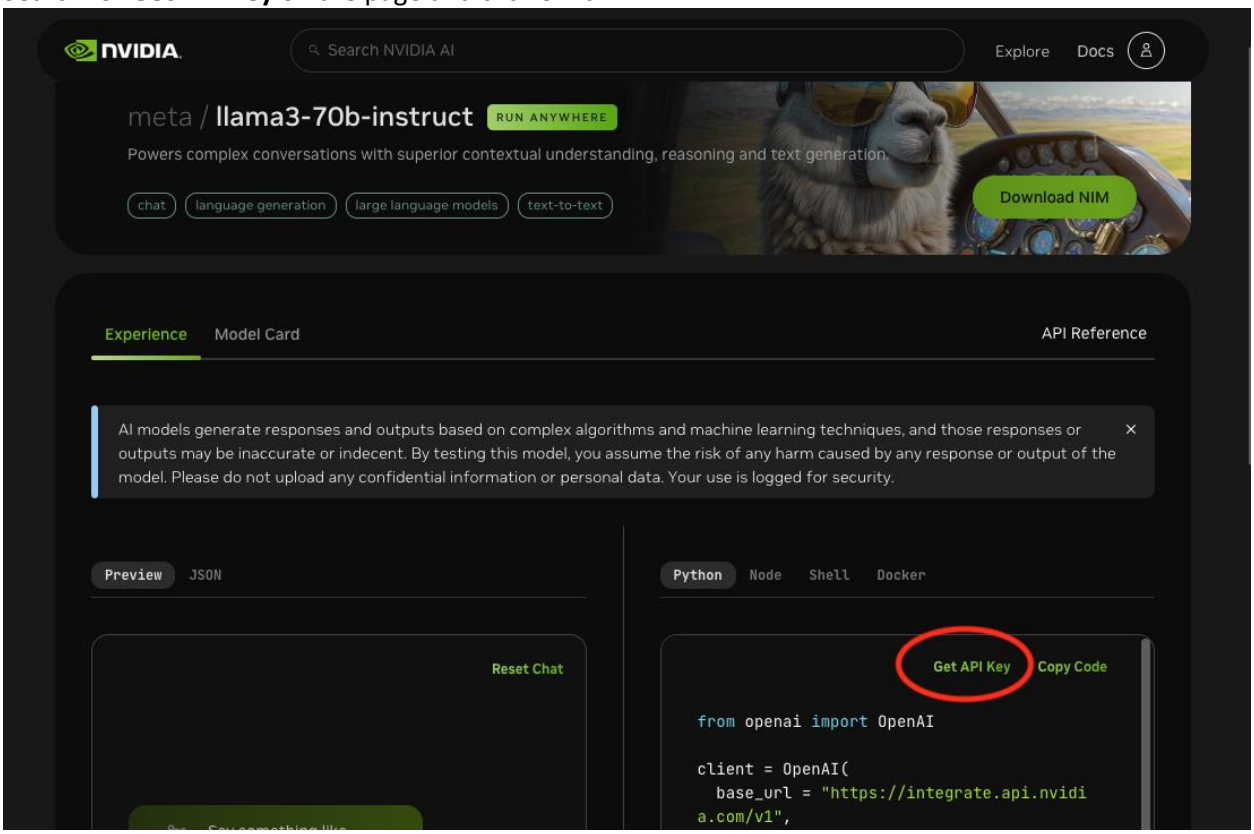
Follow the steps in https://docs.docker.com/engine/install/ubuntu/ to install the Docker Engine on Ubuntu.

Install NVIDIA Container Toolkit

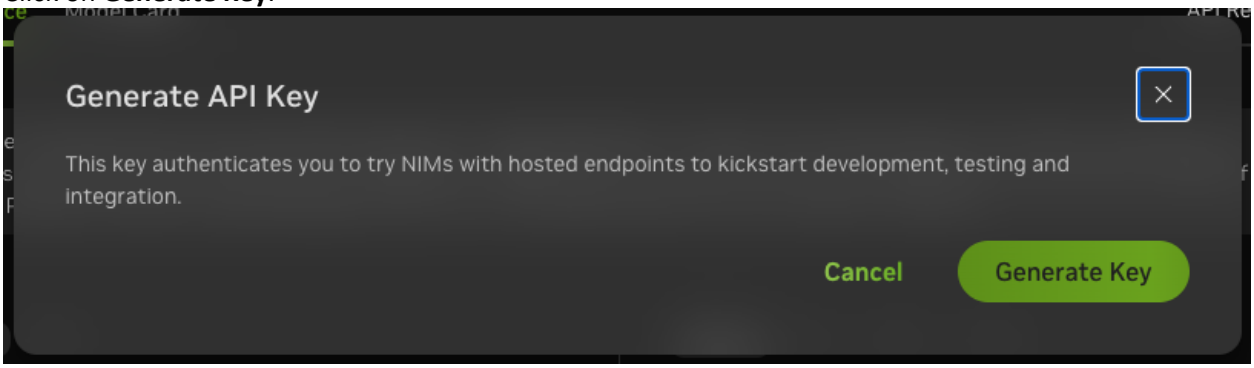
Follow the steps in https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html#installing-the-nvidia-container-toolkit to install the NVIDIA Container Toolkit

Obtain NVIDIA API Key

1. Log in to https://build.nvidia.com/explore/discover.
2. Navigate to https://build.nvidia.com/meta/llama3-70b.
3. Search for **Get API Key** on the page and click on it.



4. Click on **Generate Key**.



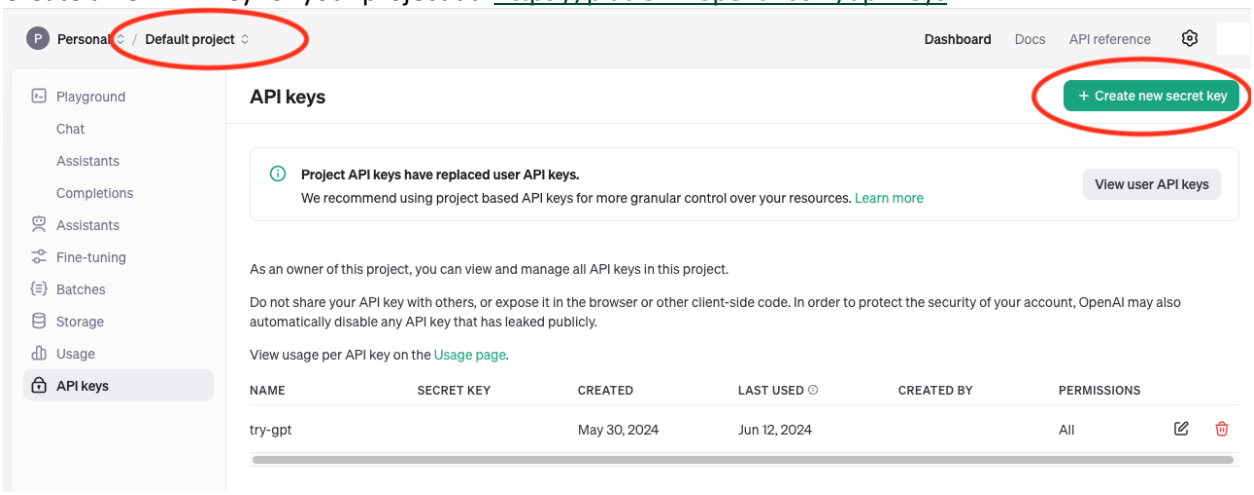
5. Store the generated API Key securely for future use.
6. Use the key to configure the `NVIDIA_API_KEY` environment used by VIA. More info in sections like Starting VIA Microservice with Minimal Configuration, and Running VIA with OpenAI compatible VLM from NVIDIA (NIM).

Obtain NGC API Key

Follow the steps in https://docs.nvidia.com/ngc/gpu-cloud/ngc-user-guide/index.html#generating-api-key to obtain the required NGC API Key.

Obtain OpenAI API Key

- Login at: https://platform.openai.com/apps
- Select: API
- Create a new API key for your project at: https://platform.openai.com/api-keys



7. Store the generated API Key securely for future use.
8. Use the key configure the `OPENAI_API_KEY` environment used by VIA. For more info see sections like GPT-4o (OpenAI) and Running VIA with OpenAI API compatible VLM.

Download VIA Container Image

- Login to the NVIDIA Container Repository using your NGC API Key

```
docker login nvcr.io -u '$oauthtoken' -p <NGC_API_KEY>
```

- Pull the docker image

```
docker pull nvcr.io/metropolis/via-dp/via-engine:2.0-dp
```

Running VIA Microservice

Starting VIA Microservice with Minimal Configuration

GPT-4o (OpenAI) - Default

- Run the following commands in a terminal:

```
export BACKEND_PORT=8000
export FRONTEND_PORT=9000
export NVIDIA_API_KEY=<YOUR-NVIDIA-API-KEY>
export OPENAI_API_KEY=<YOUR-OPENAI-API-KEY>

docker run --rm -it --ipc=host --ulimit memlock=-1 \
       --ulimit stack=67108864 --tmpfs /tmp:exec --name via-server \
       --gpus '"device=all"' \
       -p $FRONTEND_PORT:$FRONTEND_PORT \
       -p $BACKEND_PORT:$BACKEND_PORT \
       -e BACKEND_PORT=$BACKEND_PORT \
       -e FRONTEND_PORT=$FRONTEND_PORT \
       -e NVIDIA_API_KEY=$NVIDIA_API_KEY \
       -e OPENAI_API_KEY=$OPENAI_API_KEY \
       -v via-hf-cache:/tmp/huggingface \
       nvcr.io/metropolis/via-dp/via-engine:2.0-dp
```

The following logs are seen after the microservice starts:

```
************************************************************
VIA Server loaded
Backend is running at http://0.0.0.0:8000
Frontend is running at http://0.0.0.0:9000
Press ctrl+C to stop
************************************************************
```

- After the microservice starts, the VIA API is available at http://<HOST_IP>:8000 and the Demo UI is available at http://<HOST_IP>:9000.

VITA-2.0 (Optional)

- Run the following commands in a terminal:

```
export BACKEND_PORT=8000
export FRONTEND_PORT=9000
export NVIDIA_API_KEY=<YOUR-NVIDIA-API-KEY>
export NGC_API_KEY=<YOUR-NGC-API-KEY>
export MODEL_PATH="ngc:nvidia/tao/vita:2.0.1"
export NGC_MODEL_CACHE=</SOME/DIR/ON/HOST>

docker run --rm -it --ipc=host --ulimit memlock=-1 \
       --ulimit stack=67108864 --tmpfs /tmp:exec --name via-server \
       --gpus '"device=all"' \
       -p $FRONTEND_PORT:$FRONTEND_PORT \
       -p $BACKEND_PORT:$BACKEND_PORT \
       -e BACKEND_PORT=$BACKEND_PORT \
       -e FRONTEND_PORT=$FRONTEND_PORT \
       -e NVIDIA_API_KEY=$NVIDIA_API_KEY \
       -e NGC_API_KEY=$NGC_API_KEY \
       -e VLM_MODEL_TO_USE=vita-2.0 \
       -v $NGC_MODEL_CACHE:/root/.via/ngc_model_cache \
       -e MODEL_PATH=$MODEL_PATH \
       -v via-hf-cache:/tmp/huggingface \
       nvcr.io/metropolis/via-dp/via-engine:2.0-dp
```

When the above commands are first run, the microservice downloads the VITA 2.0 model from NGC, generates the TensorRT-LLM engine for the model, and starts the microservice on all GPUs installed on the host. For subsequent runs, the NGC model is cached to the directory specified by `NGC_MODEL_CACHE`.

Configuration Options

The following table describes all the configuration options supported by the VIA microservice. Each configuration can be set as an environment variable during the Docker run command.

Mandatory Configuration Option	Environment Variable	Notes
Backend (API) Port	BACKEND_PORT=8000	This is a required option. The corresponding Docker port must also be exposed.

		For more information see Configuring Server Ports.
Frontend Port	FRONTEND_PORT=9000	This is a required option. The corresponding Docker port must also be exposed. For more information see Configuring Server Ports.

Optional Configuration Option	Environment Variable	Notes
Asset storage	ASSET_STORAGE_DIR=</some/path>	Required if uploaded files / assets must be persisted. See Persisting Files / Assets on Host.
Path to CA RAG config file on host	CA_RAG_CONFIG=""	Default is /opt/nvidia/via/default_config.yaml. See Using a custom CA-RAG configuration for details.
Disable CA RAG	DISABLE_CA_RAG=false	Default is false. See Disable Frontend / Guardrails / CA-RAG for details.
Disable the VIA Frontend	DISABLE_FRONTEND=false	Default is false. See Disable Frontend / Guardrails / CA-RAG for details.
Disable Guardrails	DISABLE_GUARDRAILS=false	Default is false. See Disable Frontend / Guardrails / CA-RAG for details.
Configure directory to load input videos for VIA frontend	EXAMPLE_STREAMS_DIR=""	Default is /opt/nvidia/via/streams. See Loading Sample Streams from Local Filesystem (UI)
NVIDIA API Key	NVIDIA_API_KEY=<>	This is required when using Guardrails or CA-RAG.
NGC API Key	NGC_API_KEY=""	Required if downloading models from NGC. See Downloading NGC Models for details.
NGC Model cache folder	NGC_MODEL_CACHE=""	Required if NGC download models must be persisted. See Downloading NGC Models
Model Path (Local Model / NGC)	MODEL_PATH=</model/path/dir> MODEL_PATH= ngc:<NGC_MODEL_RESOURCE_STR>	Path to load the VITA model or another custom model from. Either a local directory or NGC resource. See Loading Custom Model / VITA Model Loading Configuration / Downloading NGC Models for details.

VLM Model TRT engine path	TRT_ENGINE_PATH=""	Required only if TRT engine is at a non-standard location. Default will use the standard location inside the model path. See VITA Model Loading Configuration for details.
VLM Model preferred TRT precision	USE_TRT_INT8=false	Use int8 mode while building TRT engine (only if TRT engine does not exist). Default is false. See VITA Model Loading Configuration for details.
VLM Model batch size	VLM_BATCH_SIZE=4	Default = 4 for GPUs with GPU memory > 80GB and = 1 for others. See VITA Model Loading Configuration for details.
VLM Model to use	VLM_MODEL_TO_USE=vita-2.0	Default is `openai-compat` Options: `vita-2.0`, `openai-compat`, `custom`
Milvus Data store directory	MILVUS_DATA_DIR=""	Set if milvus data must be persisted on host. See Persisting Milvus Data on Host.

Configurations to use OpenAI compatible models

See Running VIA with OpenAI API Compatible VLM for details.

Configuration Option	Environment Variable	Notes
VIA VLM Model Name	VIA_VLM_OPENAI_MODEL_DEPLOYMENT_NAME	Mandatory
VIA VLM Model endpoint URL	VIA_VLM_ENDPOINT	Default is the OpenAI v1 endpoint= “https://api.openai.com/v1/”.
API Key to use for the VIA VLM Model	VIA_VLM_API_KEY	API Key to use for the VIA VLM Model hosted at VIA_VLM_ENDPOINT.

VIA VLM Model API version	OPENAI_API_VERSION	Optional as required for the model deployment.
VIA VLM Model API version when using Azure OpenAI	AZURE_OPENAI_API_VERSION	Optional as required for the model deployment.
OpenAI API Key	OPENAI_API_KEY	Mandatory when using model directly from OpenAI.

Selecting GPUs

The VIA microservice runs on all the GPUs that are made available to the container.

- To make all GPUs installed on the host available to the container and VIA microservice, use `--gpus '"device=all"'` with the `docker run` command.
- To use specific GPUs, use `--gpus '"device=<device-ids>"'` with the `docker run` command. For example, `--gpus '"device=2,3"'`.

For more information on how to make certain GPUs available to the container, see https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/docker-specialized.html#gpu-enumeration.

Configuring Server Ports

The VIA Backend (API) and UI Application ports can be configured using environment variables `BACKEND_PORT` and `FRONTEND_PORT` respectively. Along with the environment variables, the ports need to be exposed when starting the Docker container so that the application is accessible from outside the container.

For example:

```
docker run ... -p <BACKEND_PORT>:<BACKEND_PORT> -e BACKEND_PORT=<BACKEND_PORT> \
      -p <FRONTEND_PORT>:<FRONTEND_PORT> -e FRONTEND_PORT=<FRONTEND_PORT> ...
```

Load VITA from Local Filesystem

To load a VITA model from host file system:

- Mount the directory containing the VITA model checkpoints and the visual embedding model checkpoints in the container.

- Make sure that the VITA `config.json` file correctly points to the visual embedding model path in the container and set the mount path to the `MODEL_PATH` env variable.

For example:

```
ls <MODEL_DIR_ON_HOST>
vila-1.5-llama-3-8b-vision_tower config.json model-00001-of-00004.safetensors ...

docker run ... -v <MODEL_DIR_ON_HOST>:<MODEL_DIR_IN_CONTAINER> \
        -e MODEL_PATH=<MODEL_DIR_IN_CONTAINER> ...

# Make sure "mm_vision_tower" in <MODEL_DIR_IN_CONTAINER>/config.json points to
"<MODEL_DIR_IN_CONTAINER>/vila-1.5-llama-3-8b-vision_tower"
```

While loading the model, the VIA microservice generates the TRT-LLM engine for it if it doesn't exist already. See VITA TRT-LLM Engine Configuration for details.

VITA TRT-LLM Engine Configuration

VIA microservice builds the TRT-LLM engine for the VITA model if it doesn't exist already. By default, it selects FP16 precision, a batch size depending on GPU memory, and generates the TRT-LLM engines inside the model directory.

These can be configured using:

- env variable `VLM_BATCH_SIZE` – The GPU must be able to support the specified batch size.
- env variable `USE_TRT_INT8` – If set to `true`, INT8 precision is used to build the engine.
- env variable `TRT_ENGINE_PATH` – If set, the VIA microservice looks for a pre-existing engine in the specified path. If not found, it generates the engine at that path.

For example:

```
docker run ... -e VLM_BATCH_SIZE=4 -e USE_TRT_INT8=true \
        -v <TRT_LLM_ENGINE_DIR_ON_HOST>:<TRT_LLM_ENGINE_DIR_IN_CONTAINER> \
        -e TRT_ENGINE_PATH=<TRT_LLM_ENGINE_DIR_IN_CONTAINER> ...
```

Using a Custom CA-RAG Configuration

To use a custom CA-RAG configuration:

- Save the configuration to a file on the host, mount it in the VIA container, and set the mount path to the `CA_RAG_CONFIG` env variable.

For example:

```
docker run ... -v <CA_RAG_CONFIG_FILE_ON_HOST>:<CA_RAG_CONFIG_FILE_IN_CONTAINER> \
        -e CA_RAG_CONFIG=<CA_RAG_CONFIG_FILE_IN_CONTAINER> ...
```

Downloading NGC Models

To download NGC models:

- Locate or define the required NGC model resource string and NGC API key. Refer to Obtaining NGC API KEY for details.
- Export the NGC model resource string as the `MODEL_PATH` env variable.
- Export the API key as the `NGC_API_KEY` env variable while running the VIA container.

For example:

```
docker run ... -e MODEL_PATH=ngc:<NGC_MODEL_RESOURCE_STR> \
       -e NGC_API_KEY=<YOUR-NGC-API-KEY> ...
```

After the model is downloaded, the VIA microservice proceeds to generate the TRT-LLM engine for it. See the VITA TRT-LLM Engine Configuration for details.

The VIA microservice avoids re-downloading NGC models if the model is found in its cache. NGC models are downloaded to the container storage by default and are lost if the container is restarted.

To persist NGC downloaded models, mount a host directory or Docker volume in the container, and set `NGC_MODEL_CACHE` env variable to the mount path.

For example:

```
docker run ... -v <NGC_MODEL_DIR_ON_HOST>:/root/.via/ngc_model_cache ...
```

Loading Sample Streams from Local Filesystem (UI)

To load a sample stream from a local filesystem as examples in the UI, mount a host directory in the container and set `EXAMPLE_STREAMS_DIR` to the mount path.

For example:

```
docker run ... -v <STREAMS_DIR_ON_HOST>:<STREAMS_DIR_IN_CONTAINER> \
       -e EXAMPLE_STREAMS_DIR=<STREAMS_DIR_IN_CONTAINER> ...
```

NOTE: The directory must contain only valid video files. The presence of non-video files or directories may lead to errors.

Load Custom Model

To load a custom model:

- Mount the directory containing the `inference.py` file and the optional model files in the container.
- Set the `MODEL_PATH` env variable to the mount path. MODEL_PATH should point to the directory containing the `inference.py` file in the container.
- Set `VLM_MODEL_TO_USE=custom`

For example:

```
ls <MODEL_DIR_ON_HOST>
inference.py <model-checkpoints-dir>

docker run ... -v <MODEL_DIR_ON_HOST>:<MODEL_DIR_IN_CONTAINER> \
        -e MODEL_PATH=<MODEL_DIR_IN_CONTAINER> -e VLM_MODEL_TO_USE=custom ...
```

Persisting Files / Assets on Host

By default, VIA microservice stores the uploaded files and any added assets in the container local storage. To persist the assets, mount a host directory in the container and set `ASSET_STORAGE_DIR` to the mount path.

For example:

```
docker run ... -v <ASSET_DIR_ON_HOST>:<ASSET_DIR_IN_CONTAINER> \
        -e ASSET_STORAGE_DIR=<ASSET_DIR_IN_CONTAINER> ...
```

Persisting Milvus Data on Host

By default, the Milvus Server that is started inside the VIA container, stores data inside the container. To persist Milvus data on host storage:

- Mount a host directory and Docker volume in the container.
- Set `MILVUS_DATA_DIR` env. variable to its path.

For example:

```
docker run ... -v <MILVUS_DATA_DIR_ON_HOST>:<MILVUS_DATA_DIR_IN_CONTAINER> \
        -e MIVLUS_DATA_DIR=<MILVUS_DATA_DIR_IN_CONTAINER> ...
```

Disable Frontend, Guardrails, and CA-RAG

VIA microservice allows disabling some parts of the microservice. This can be done by setting env variables `DISABLE_GUARDRAILS`, `DISABLE_CA_RAG`, `DISABLE_FRONTEND` to `true`.

For example:

```
docker run ... -e DISABLE_GUARDRAILS=true \
        -e DISABLE_CA_RAG=true -e DISABLE_FRONTEND=true ...
```

Running VIA with OpenAI API Compatible VLM

To use an OpenAI API Compatible VLM in VIA, set env variable `VLM_MODEL_TO_USE` to `openai-compat` along with authentication and model parameters as defined below:

Deployment	Parameter Type	Required Environments
OpenAI Direct	Authentication	OPENAI_API_KEY=<key>
	Model	VIA_VLM_OPENAI_MODEL_DEPLOYMENT_NAME=<deployment-name>
	Example: GPT4o	VIA_VLM_OPENAI_MODEL_DEPLOYMENT_NAME=gpt-4o
	Example: GPT4v Turbo	VIA_VLM_OPENAI_MODEL_DEPLOYMENT_NAME=gpt-4-turbo-2024-04-09

For example:

Running GPT-4o with OpenAI:

```
docker run ... -e OPENAI_API_KEY=<OPENAI_API_KEY> \
        -e VIA_VLM_OPENAI_MODEL_DEPLOYMENT_NAME=gpt-4o \
        -e VLM_MODEL_TO_USE=openai-compat ...
```

Model Details

VIA uses the following models

VIA VLM Models:

GPT-4o and GPT4-v Turbo

VIA offers support to use OpenAI models like GPT-4o and GPT-4v turbo as VLM. GPT-4o is the VLM configured by default.

See Running VIA with OpenAI compatible VLM for how to set up VIA to use these models.

NVIDIA VITA

NVIDIA VITA is Video Language Model (VLM) which can be deployed locally. VITA (Vision Temporal Assistant) is the LITA model that uses the VILA as encoder. More details on Language Instructed Temporal Assistant (LITA) are available at https://arxiv.org/pdf/2403.19046v1 and for VILA at https://arxiv.org/pdf/2312.07533v4 and https://github.com/NVlabs/VILA.

VIA microservice can be optionally configured to use NVIDIA VITA instead of GPT-4o.

Custom VLM Models

VIA supports integrating custom VLM models in addition to the VLM models mentioned above. Refer to Integrating Custom VLM models.

VIA CA-RAG Models:

LLaMA 3 70b Instruct

The LLaMA 3 70b Instruct model is used for Guardrails and by CA-RAG. The model is by default used as NVIDIA Hosted NIM. You can use a locally deployed NIM as well as other Instruction tuned LLMs, if they support OpenAI APIs.

Python CLI Client

Pre-requisites

A reference Python CLI client is provided along with the VIA microservice. The client internally calls the REST APIs exposed by the VIA microservice.

The client is added to the VIA container image at `/opt/nvidia/via/via_client_cli.py`. It is also available on the git repo at https://github.com/NVIDIA-AI-IOT/via-engine/blob/main/via_client_cli.py.

The Python package dependencies for the CLI client can be installed using:

```
pip3 install tabulate tqdm sseclient-py requests
```

The CLI client can be executed by running:

```
python3 via_client_cli.py <command> <args> [--backend <VIA_API_URL>] [--print-curl-command]
```

By default, the client assumes that the VIA API server is running at http://localhost:8000. This can be configured by passing the argument `--backend <VIA_API_URL>` as shown above.

The CLI client also provides an option to print the curl command for any operation. This can be done by passing the `--print-curl-command` argument to the client.

To get a list of all supported commands and options supported by each command run:

```
python3 via_client_cli.py -h

python3 via_client_cli.py <command> -h
```

Files Commands

The following section describes each of the commands in detail.

Add File

Calls `POST /files` internally. Uploads or adds a file as path. Prints the file id and other details.

Reference:

```
via_client_cli.py add-file [-h] [--add-as-path] [--backend BACKEND] [--print-curl-command]
file
```

Example for uploading a file:

```
python3 via_client_cli.py add-file video.mp4
```

Example for adding a file as path (This requires the file path to be accessible inside the container):

```
python3 via_client_cli.py add-file --add-as-path /media/video.mp4
```

List Files

Calls `GET /files` internally. Prints the list of files added to the server and their details in a tabular format.

Reference:

```
via_client_cli.py list-files [-h] [--backend BACKEND] [--print-curl-command]
```

Example:

```
python3 via_client_cli.py list-files
```

Get File Details

Calls `GET /files/{id}` internally. Prints the details of the file.

Reference:

```
via_client_cli.py file-info [-h] [--backend BACKEND] [--print-curl-command] file_id
```

Example:

```
python3 via_client_cli.py file-info 7ce1127a-2009-4bfa-bdf8-efa9e1f37fa4
```

Get File Contents

Calls `GET /files/{id}/content` internally. Saves the content to a new file.

Reference:

```
via_client_cli.py file-content [-h] [--backend BACKEND] [--print-curl-command] file_id
```

Example:

```
python3 via_client_cli.py file-content 7ce1127a-2009-4bfa-bdf8-efa9e1f37fa4
```

Delete File

Calls `DELETE /files/{id}` internally. Prints the delete status and file details.

Reference:

```
via_client_cli.py delete-file [-h] [--backend BACKEND] [--print-curl-command] file_id
```

Example:

```
python3 via_client_cli.py delete-file 7ce1127a-2009-4bfa-bdf8-efa9e1f37fa4
```

Live Stream Commands

Add Live Stream

Calls `POST /live-stream` internally. Prints the live-stream id if it is added successfully.

Reference:

```
via_client_cli.py add-live-stream [-h] [--description DESCRIPTION]
                                      [--username USERNAME] [--password PASSWORD]
                                      [--backend BACKEND] [--print-curl-command]
                                      live_stream_url
```

Example:

```
python3 via_client_cli.py add-live-stream --description “Some live stream description” \
     rtsp://192.168.1.100:8554/video/media1
```

List Live Streams

Calls `GET /live-stream` internally. Prints the list of live-streams and their details in a tabular format.

Reference:

```
via_client_cli.py list-live-streams [-h] [--backend BACKEND] [--print-curl-command]
```

Example:

```
python3 via_client_cli.py list-live-streams
```

Delete Live Stream

Calls `DELETE /live-stream/{id}` internally. Prints the status confirming deletion of the live stream

Reference:

```
via_client_cli.py delete-live-stream [-h] [--backend BACKEND] [--print-curl-command]
                                     video_id
```

Example:

```
python3 via_client_cli.py delete-live-stream ea071500-3a47-4e6f-87da-1bc796075344
```

Models Commands

List Models

Calls `GET /models` internally. Prints the list of models loaded by the server and their details in a tabular format.

Reference:

```
via_client_cli.py list-models [-h] [--backend BACKEND] [--print-curl-command]
```

Example:

```
python3 via_client_cli.py list-models
```

Summarization Command

Calls `POST /summarize` internally. Triggers summarization on a file or live-stream and blocks it until summarization is complete or you interrupt the process.

The command allows some configurable parameters with the summarize request.

For files, results are available after the entire file is summarized. The command then prints the results.

For live-streams, results are periodically available. The period depends on the `chunk_duration` and `summary_duration`. Interrupting the summarize command does not stop the summarization on the server side. You can re-connect to the live-stream by re-running the summarize command with the same `id` as the live-stream.

For more details on each argument, see the help command and the VIA API reference.

Reference:

```
via_client_cli.py summarize [-h] --id ID --model MODEL
                            [--stream]
                            [--chunk-duration CHUNK_DURATION]
                            [--chunk-overlap-duration CHUNK_OVERLAP_DURATION]
                            [--summary-duration SUMMARY_DURATION]
                            [--prompt PROMPT]
                            [--file-start-offset FILE_START_OFFSET]
                            [--file-end-offset FILE_END_OFFSET]
                            [--model-temperature MODEL_TEMPERATURE]
                            [--model-top-p MODEL_TOP_P]
                            [--model-top-k MODEL_TOP_K]
                            [--model-max-tokens MODEL_MAX_TOKENS]
                            [--model-seed MODEL_SEED]
                            [--response-format {json_object,text}]
                            [--backend BACKEND] [--print-curl-command]
```

Example:

```
python3 via_client_cli.py summarize \
        --id ea071500-3a47-4e6f-87da-1bc796075344 \
        --model vita-2.0 \
        --chunk-duration 60 \
        --stream \
        --prompt “Write a dense caption about the video containing events like ...” \
        --model-temperature 0.8
```

Server Health and Metrics Commands

Server Health Check

Calls `GET /health/ready` internally. Checks the response status code and prints the server health status.

Reference:

```
via_client_cli.py server-health-check [-h] [--backend BACKEND] [--print-curl-command]
```

Example:

```
python3 via_client_cli.py server-health-check
```

Server Metrics

Calls `GET /metrics` internally. Prints the server metrics. The metrics are in Prometheus format.

Reference:

```
via_client_cli.py server-metrics [-h] [--backend BACKEND] [--print-curl-command]
```

Example:

```
python3 via_client_cli.py server-metrics
```

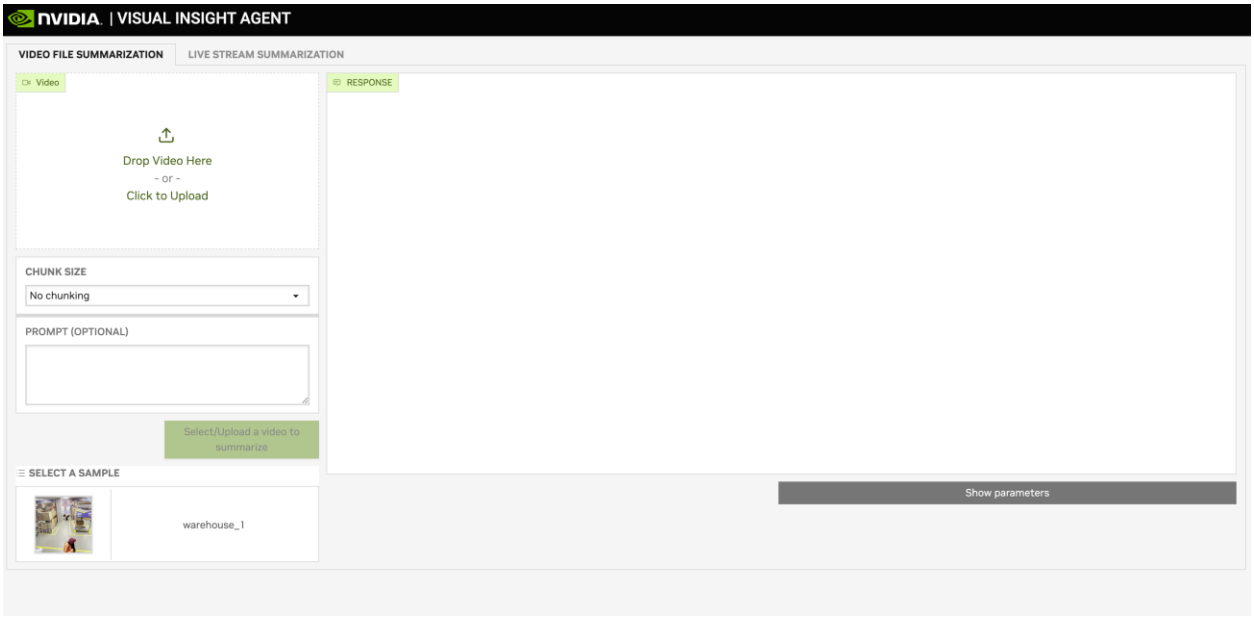
UI Application

A sample UI application based on Gradio is provided along with VIA. It supports file and live stream summarization.

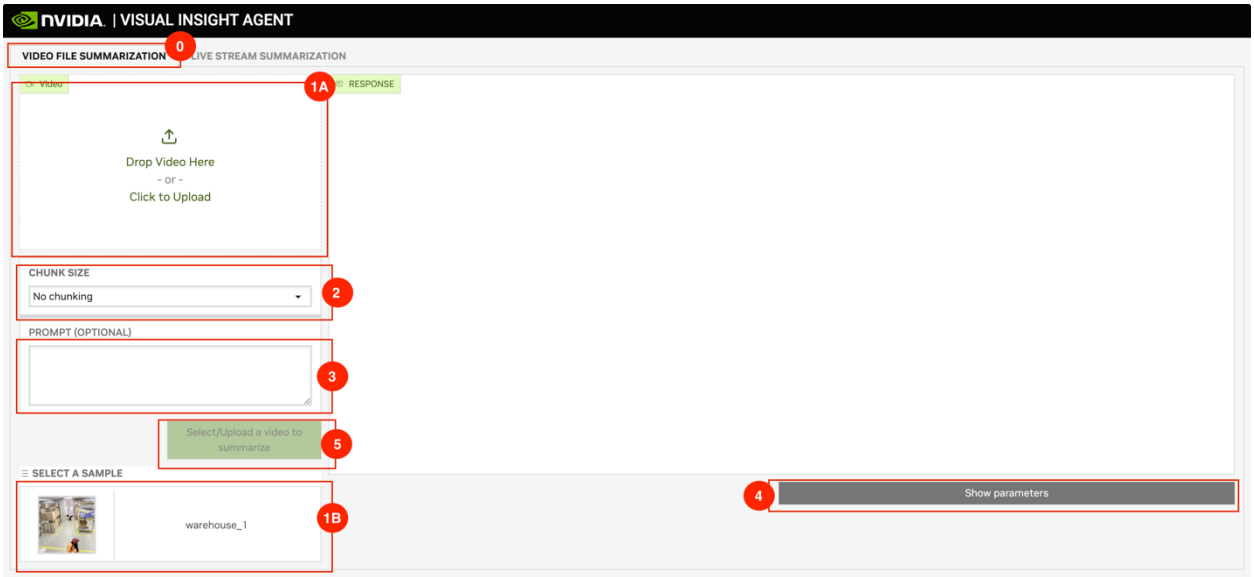
The VIA container logs show the port on which the UI application server is running. The UI can be accessed by navigating to `http://<VIA_HOST_IP:<FRONTEND_PORT>.`

File Summarization

The following image shows the file summarization page of the UI.



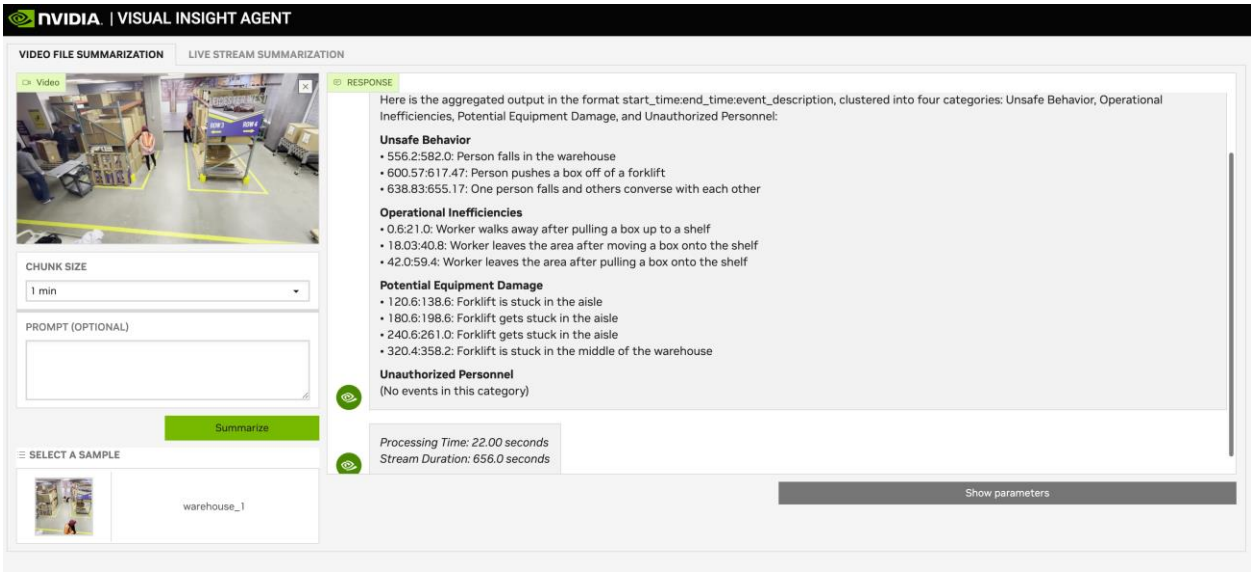
To start file summarization, follow the steps mentioned in the image and in the text below:



0. Make sure the “Video File Summarization” tab is selected.
1. Either upload a file or choose a preloaded example.
2. Select the chunk size from the drop down list.
3. Set a prompt (Optional).
4. Adjust the VLM parameters by clicking on Show Parameters and modifying the available parameters (Optional).
5. Click on the Summarize button to start summarization. The button gets enabled after a video is selected or uploaded.

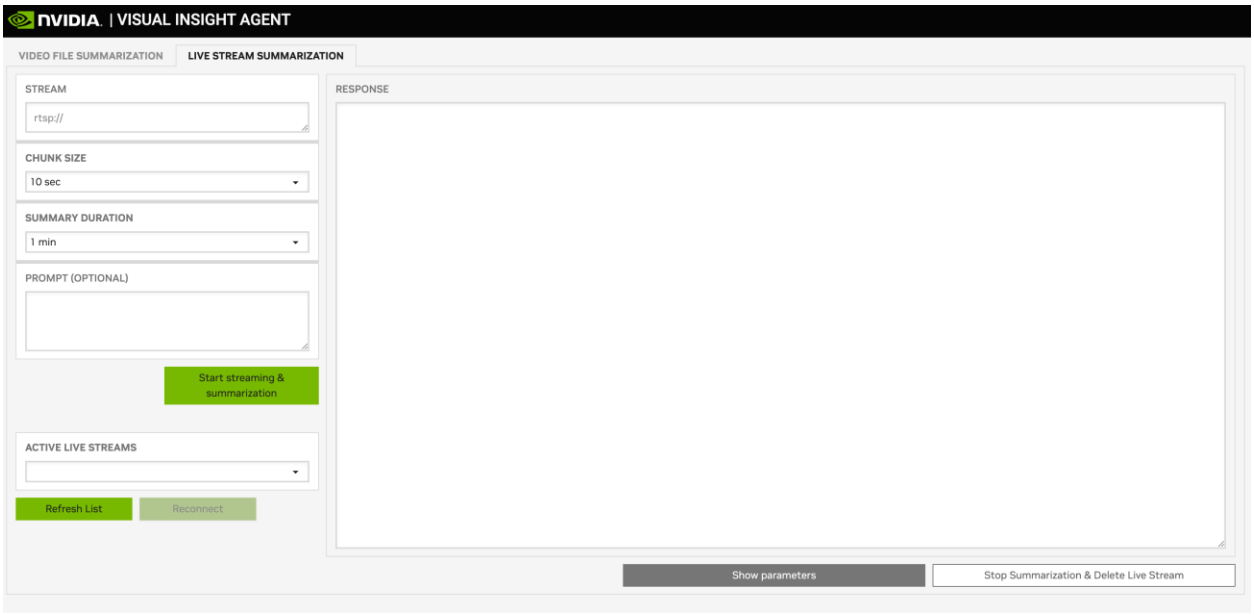
The summarization takes from a few seconds to minutes, depending on various factors including video length, chunk size, prompt, VLM model, GPUs installed on the host.

After the processing is finished, the response shows the video summary.

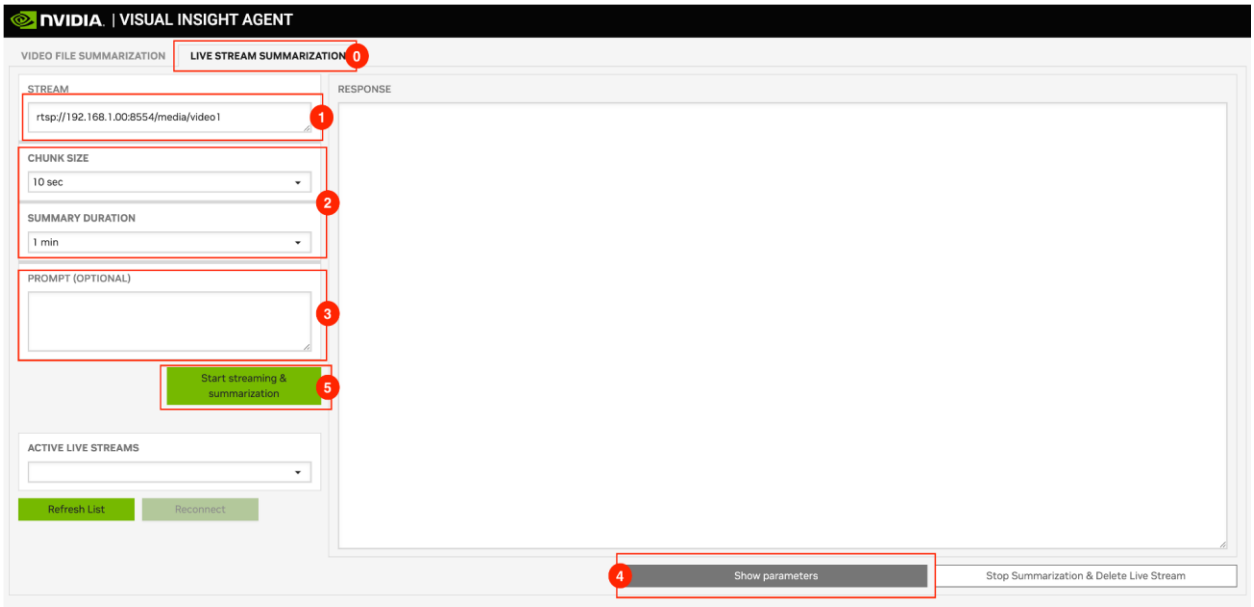


Live Stream Summarization

The following image shows the live stream summarization page of the UI.



To start file summarization, follow the steps mentioned in the image and text below:

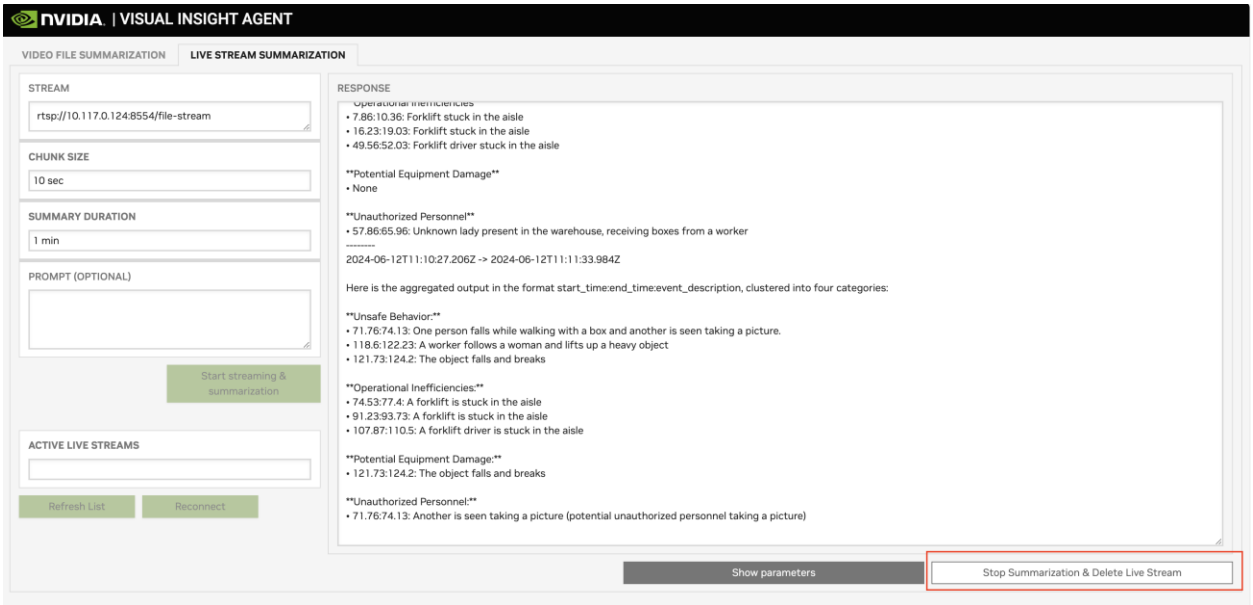


0. Make sure the “Live Stream Summarization” tab is selected.
1. Enter a valid RTSP Live Stream URL.
2. Select the chunk size & summary duration from the drop down list.
3. Set a prompt (Optional).
4. Adjust the VLM parameters by clicking on Show Parameters and modifying the available parameters (Optional).
5. Click on Start Streaming & Summarize button to start summarization. The button gets enabled once RTSP URL is entered.

After the server starts processing the stream, the UI displays a message “Waiting for first summary”. The first summary is available after approximately summary duration + a few seconds depending on the chunk duration, summary duration, model, and GPU being used.

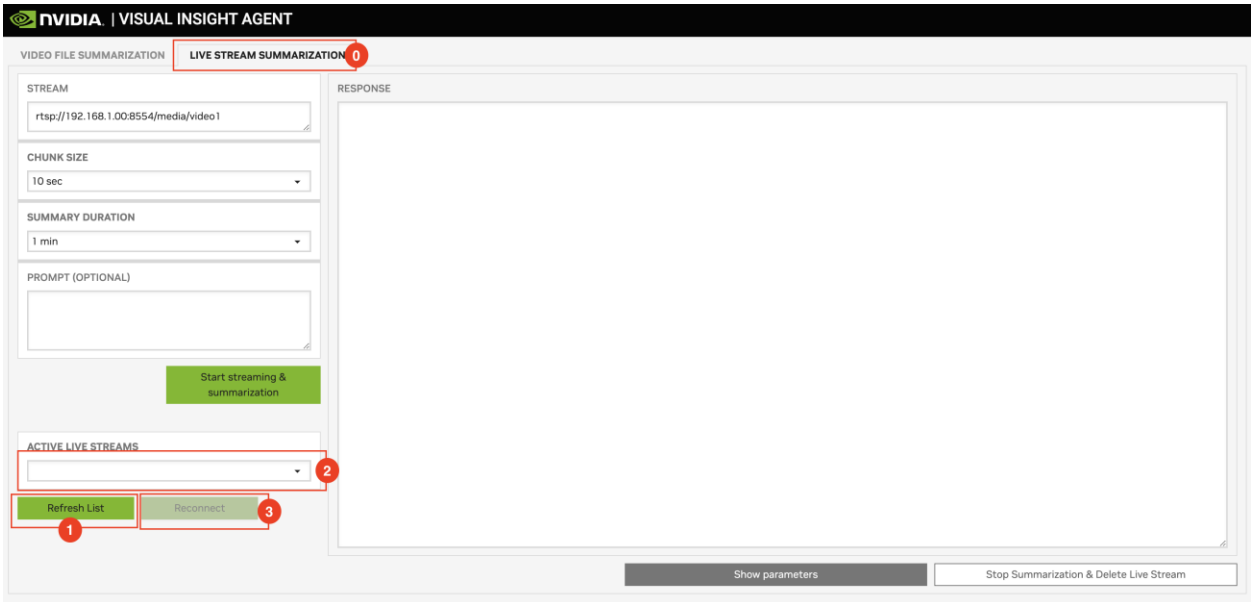
Summaries for consecutive summary duration periods are appended one below the other. Newest one being at the bottom.

Live Stream Summarization can be stopped by clicking on “Stop Summarization & Delete Live Stream”.



Reconnecting to Live Stream Summarization

For live streams, even if the client disconnects, summarization keeps happening in the VIA server. To reconnect to the live stream and get latest updates since being disconnected, follow the steps mentioned in the image and text below:



0. Make sure the “Live Stream Summarization” tab is selected.
1. Click on “Refresh List” button.
2. Once the list is refreshed, select the active stream from the drop down.
3. Click on Reconnect. The button will be enabled once a stream is selected from the drop-down.

Context-Aware RAG

VIA Context Aware RAG provides the framework for performing retrieval on the video data based on the context and generating the context to answer user's questions. Currently, we support summarization.

CA RAG includes the following:

- **Handlers** – A handler is an abstraction of the tools available to CA-RAG. Implementing various handlers enables various use cases.
- **Context Manager** – CM uses the handlers and performs a specific operation. For example, CA RAG uses the LLM handler to implement the summarize operation.

Handlers

LLM Handler

This is a handler to interface an LLM. Currently, we have implemented `LLMNVIDIA`, which connects to an LLM NIM and provides access to multiple LLMs. Other LLM handlers can be implemented, such as a self-hosted LLM or third-party LLMs.

DB Handler

This handler interfaces with databases. Currently, we have `MilvusDBHandler`, which stores and fetches data from MilvusDB. This DB is where video captions are stored.

More handlers can be added, including notification handlers, and a highlight clip generation handler.

Context Manager

Context Manager uses the various handlers to perform tasks such as Summarization, etc.

Currently, CM uses the `DBHandler` and `LLMHandler` to summarize a video.

Summarization

Three methods of summarization are supported **stuff**, **batch,** or **refine**:

- **stuff** - All the documents are appended, and a summary is generated. Recommended for a small number of documents.
- **refine** - This is a recursive method where a summary is generated using the previous summary and a new caption. Recommended for streaming captions.
- **batch** - This is a two-step summarization. Step 1 is creating batches of batch_size and generating the summaries for these batches. Step 2 is combining the batch summaries using a second prompt, summary_aggregation. Recommended for large documents.

Usage

```
from via_ctx_rag.context_manager.context_manager import ContextManager
from via_ctx_rag.llm.llm_handler import LLMNVIDIA

ctx_mgr = ContextManager(config)
ctx_mgr.add_handlers([
        LLMNVIDIA(
            model=ctx_mgr.llm_params["model"],
            api_key=api_key,
            max_tokens=ctx_mgr.llm_params["max_tokens"]),
        ]
    )
# documents: list(dict)
# [
#    {
#       "start_pts": 0,
#       "end_pts": 100,
#       "text": "caption"
#    },
#    ...
#]
summary = ctx_mgr.summarize(documents)

print(summary)
```

VIA Microservice Customization

Integrating Custom VLM models

VIA supports integrating custom VLM models. Depending on the model to be integrated, either some configurations might need to be updated or interface code would have to be implemented. The model can ONLY be selected at initialization time.

Custom VITA Checkpoint

To use custom VITA checkpoint but the same architecture as the publicly available VITA model, refer to Load VITA from Local Filesystem / Downloading NGC models.

If the checkpoint is not compatible, refer to Other Custom Models.

OpenAI Compatible REST API

If the VLM model provides an OpenAI compatible REST API, refer to Running VIA with OpenAI compatible VLM.

Other Custom Models

VIA allows you to drop in your own models to the model directory by providing the pre-trained weight of the model and implementing an interface to bridge to the VIA pipeline.

The interface includes an `inference.py` file and a `manifest.yaml`.

In the `inference.py`, users must define a class named `Inference` with the following two methods:

```
def get_embeddings(self, tensor:torch.tensor) -> tensor:torch.tensor:
    # Generate video embeddings for the chunk / file.
    # Do not implement if explicit video embeddings are not supported by model
    return tensor

def generate(self, prompt:str, input:torch.tensor, configs:Dict):
    # Generate summary string from the input prompt and frame/embedding input.
    # configs contains VLM generation parameters like
    # max_new_tokens, seed, top_p, top_k, temperature
```

```
    return summary
```

The optional `get_embeddings` method is used to generate embeddings for a given video clip wrapped in a TCHW tensor and must be removed if the model doesn't support the feature.

The generate method is used to generate the text summary based on the given prompt and the video clip wrapped in a TCHW tensor.

Both models that need to be executed locally on the system or models with REST APIs can be supported using this method.

Some examples are included inside the container at `/opt/nvidia/via/via-engine/models/custom/samples` and on GitHub at https://github.com/NVIDIA-AI-IOT/via-engine/tree/main/via-engine/models/custom/samples. Examples include models `fuyu8b`, `neva` and `phi3v`.

Configurable Parameters

See Configuration Options for the list of initialization time parameters and their details.

At runtime, the `summarize` API supports the following parameters. Refer to the API schema for details:

- temperature
- seed
- top_p
- max_tokens
- top_k
- chunk_duration
- chunk_overlap_duration (File only)
- summary_duration (Live stream only)

NOTE: Lower chunk_duration may result in better accuracy but will lead to a higher number of chunks getting created and thus requiring more time to summarize the video. Typically for an hour-long video, start with 60 second chunk duration and then adjust the chunk duration based on accuracy and latency requirements.

Tuning Prompts

VIA has configured for the warehouse usecase by default. However, VLM prompts needs to be tuned for specific use cases or video content like sports, medical vision, retail. The prompt should include specific events that need to be found out. The corresponding CA-RAG configs also need update.

Warehouse prompt configuration example:

VIA prompt configuration can be changed with the CA_RAG_CONFIG environment variable. More details in section: Using a custom CA-RAG configuration.

See the sample config file for the warehouse use case in the VIA source at `via-engine/config/config.yaml`.

Config.yaml has the following three prompts:

Prompt Type	Example Prompt	Guidelines
caption	"Write a concise and clear dense caption for the provided warehouse video, focusing on irregular or hazardous events such as boxes falling, workers not wearing PPE, workers falling, workers taking photographs, workers chitchatting, forklift stuck, etc. Start and end each sentence with a time stamp."	1) This is the prompt to VLM. 2) Make sure you provide keywords necessary to aid image/video understanding. 3) Call out the types of events you want the VLM to detect. Example: Anomaly like person not wearing PPE. 4) This prompt can be updated using VIA Gradio UI: "PROMPT (OPTIONAL)" field. See: UI application for screenshots.
caption_ summarization	"You should summarize the following events of a warehouse in the format start_time:end_time:caption. If during a time segment only regular activities happen, then ignore them, else note any irregular activities in detail. The output should be bullet points in the format start_time:end_time: detailed_event_description. Don't return anything else except the bullet points."	1) This prompt is used by CA RAG to summarize captions generated by VLM. 2) This is the first step in a two-step summarization task. 3) Change it according to your use case.
summary_ aggregation	"You are a warehouse monitoring system. Given the caption in the form start_time:end_time: caption, Aggregate the following captions in the format start_time:end_time:event_description. The output should only contain bullet points.	1) This prompt is used by CA-RAG to generate the final summary. 2) Change it according to your use case.

	Cluster the output into Unsafe Behavior, Operational Inefficiencies, Potential Equipment Damage and Unauthorized Personnel."	
caption (prompt for JSON output)	“Find out all the irregular or hazardous events such as boxes falling, workers not wearing PPE, workers falling, workers taking photographs, workers chitchatting, forklift stuck, etc. Fill the following JSON format with the event information: { "all_events": [{"event": "<event caption>", "start_time": <start time of event>, "end_time": <end time of the event>}]}. Reply only with JSON output.”	1) This is the prompt to the VLM. 2) You can change the JSON format to suit your use case.

NOTE: By default, the VIA API only allows `prompt` to be configured as part of the `summarize` API. caption_summarization and summary_aggregation prompts are configured using the CA-RAG configuration file.

To also allow caption_summarization and summary_aggregation prompts to be configured as part of the `summarize` API, set env. variable `VIA_DEV_API=1` while running the VIA container.

```
docker run ... -e VIA_DEV_API=1 ...
```

Accessing Milvus Vector DB

VIA uses Milvus vector DB to store the intermediate VLM responses per chunk before aggregating and summarizing the responses using CA-RAG.

VIA starts a Milvus vector DB instance inside the container. You can add the following arguments to the Docker run command to expose the Milvus DB server and access it from outside the container:

```
docker run ... -p 19530:19530 -e MILVUS_DB_PORT=19530 ...
```

You can use standard Milvus tools like `milvus_cli` / `Milvus Python SDK` to interact with the milvus DB at `<HOST_IP>:19530`.

VIA stores per chunk metadata and per chunk VLM response in the vector DB. The VLM response is stored as a string as is and it is not parsed or stored as structured as data. As part of metadata, it stores the start / end times of the chunk, chunk index among other things. The final aggregated summarization response from CA-RAG is not stored.

The video embeddings of the chunks are not stored in the vector DB. Instead, they are stored in the asset storage directory. You can refer to the `EmbeddingHelper` module inside the via-engine for more information and code to retrieve the embeddings. To persist the asset storage directory and mount a host directory as asset storage directory refer to Persisting Files / Assets on Host.

NOTE: All data in the DB is dropped before every summary request.

Custom Post-Processing Functions

The output of VLM is stored in a Milvus vector DB. To implement custom post-processing functions, you can connect to the Milvus vector DB and use the information stored in it. For details refer to Accessing Milvus Vector DB.

Another option is to update the CA-RAG implementation as required.

Tuning Guardrails

VIA microservice has guardrails enabled by default and provides some default guardrails configuration. VIA uses NVIDIA NeMo Guardrails to provide this functionality.

The guardrails configuration is located at `/opt/nvidia/via/guardrails_config` inside the container and at https://github.com/NVIDIA-AI-IOT/via-engine/tree/main/guardrails_config.

To disable guardrails, refer to Disabling Guardrails.

To modify the guardrails configuration, copy the `guardrails_config` directory to the host, make modifications as required and then mount the modified config at `/opt/nvidia/via/guardrails_config` in the container.

For example:

```
docker run ... -v <modified_guardrails_config_dir>:/opt/nvidia/via/guardrails_config ...
```

Using Locally Deployed LLM NIM instead of NVIDIA Hosted LLM NIM

Deploy a Chat Based NIM Locally

Follow instructions at https://docs.nvidia.com/nim/large-language-models/latest/getting-started.html#launch-nvidia-nim-for-llms to deploy a LLM NIM locally.

CA-RAG and Guardrails: Using local NIM (llama3-8b)

VIA uses NVIDIA Hosted `LLaMA3-70-Instruct` NIM for Guardrails and CA-RAG by default. This can be replaced by a locally deployed NIM.

To use the locally deployed NIM:

1) Follow Steps to deploy local NIM and deploy: llama3-8b
2) Say its deployed at:
   http://<HOST-IP>:28000/v1
3) **Configure CA-RAG:**

   Modify the CA-RAG config to use the model: “meta/llama3-8b-instruct” instead of the default “meta/llama3-70b-instruct” model

   To do this, update the `summarization/llm` in config.yaml. Example (changes in bold):

```
summarization:
  enable: true
  method: "batch"
  llm:
    base_url: "http://<HOST-IP>:28000/v1"
    model: "meta/llama3-8b-instruct"
    max_tokens: 1024
    temperature: 0.5
    top_p: 1
  params:
    batch_size: 5
    batch_max_concurrency: 20
  prompts:
    caption: "caption prompt”
    caption_summarization: "caption summarization prompt"
    summary_aggregation: "summary aggregation prompt"
```

4) **Configure Guardrails:**
   Modify guardrails config to use the model: “meta/llama3-8b-instruct” instead of the default “meta/llama3-70b-instruct” model.
   To do this, follow Tuning Guardrails.
   Example guardrails_config/config.yml update for the models section:

```
models:
  - type: main
    engine: nvidia_ai_endpoints
    model: meta/llama3-8b-instruct
    parameters:
      base_url: "http://<HOST-IP>:28000/v1"
```

VIA Source Code

The VIA engine source code is hosted at https://github.com/NVIDIA-AI-IOT/via-engine.

The source code in the repository, including any modifications, can be executed in the VIA container image by mounting the repository root directory at `/opt/nvidia/via` in the container.

For example:

```
git clone git@github.com:NVIDIA-AI-IOT/via-engine.git /home/ubuntu/via-engine

docker run ... -v /home/ubuntu/via-engine:/opt/nvidia/via ...
```

CA-RAG Source Code

The Context Aware RAG source code is hosted at https://github.com/NVIDIA-AI-IOT/via-ctx-rag.

The source code in the repository, including any modifications, can be executed in the VIA container image by building a wheel using:

```
$ poetry build
```

Which generates a wheel file at:

```
$ dist/context_aware_rag-0.1.0-py3-none-any.whl
```

Copy the wheel file to the container and install the wheel file in the container using:

```
in-the-container$ pip uninstall context_aware_rag -y && pip install /path/to/
context_aware_rag-0.1.0-py3-none-any.whl
```

Known Issues:

1. For custom queries, Aggregation is not supported.
2. Models are trained on specific data/use cases so if tested on other inputs then it might give incorrect results.
3. VLM Model accuracy: Sometimes time stamps returned are not accurate. Also, it can hallucinate for certain questions. **Prompt tuning might be required**.
4. Summarization accuracy: Summarization accuracy is heavily dependent on VLM accuracy. Also, the default CA-RAG configs have been tuned for a specific use case. User can try tuning the CA-RAG config as required.
5. Guardrails sometimes fails to load. Check if the NVIDIA_API_KEY is correct. If the API Key is correct, retry after some time.
6. Following harmless warnings might be seen during VIA microservice execution. These can be safely ignored.
    a. *GLib (gthread-posix.c): Unexpected error from C library during 'pthread_setspecific': Invalid argument. Aborting*
    b. *Warning: gst-resource-error-quark: Could not read from resource …*
    Could not receive any UDP packets …
    c. *Warning: gst-stream-error-quark: No decoder available for type 'audio/mpeg, …*
    d. *GStreamer-WARNING **: 12:58:07.600: Failed to load plugin …*
7. If Gradio UI is busy with file summarization, other operations might get stuck till the file summarization is complete. This includes other browser tabs connected to the same VIA instance and launching new UI instances for the VIA instance.
8. Guardrails might not reject some prompts that are expected to be rejected. This could be because the prompt might be relevant in other contexts as well as topics in the prompt might not be configured to be rejected. You can try tuning the guardrails configuration if required.
9. If prompt is not relevant to the video content, then VIA summary might contain blank or none response.
10. OpenAI connection errors or 429 (too many requests) errors might be seen sometimes if too many requests are sent to GPT-4v / GPT-4o VLMs. If multiple GPUs are being used, try using reduced number of GPUs using --device argument in the docker run command. VIA sends one OpenAI request per GPU in parallel for video chunk inferencing. It can also be due to lower TPM/RPM limits associated with the OpenAI account.
11. CA-RAG Summarization sometimes fails with model context length exceeded error (*This model's maximum context length is 8192 tokens. However, you requested …. Please reduce the length of the messages or completion.*). You can try with lower number of chunks (higher chunk size).

12. CA-RAG Summarization might show a truncated summary response. This is due to the max_tokens. Try increasing the number in the CA-RAG config file.
13. Rarely kernel error is seen for back to back summarization requests on certain A100 systems. If such an error is encountered, the system needs to be restarted.
14. Removal of live streams when using GPT models takes a long time / UI seems to be stuck. This happens when GPT response time is more than the chunk size and the VLM pipeline cannot keep up with the live stream FPS. Try increasing the chunk size to more than the GPT response time.
15. Model outputs can vary if the order of the chunks in a batch changes when using batch size > 1.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

Arm

Arm, AMBA, and Arm Powered are registered trademarks of Arm Limited. Cortex, MPCore, and Mali are trademarks of Arm Limited. All other brands or product names are the property of their respective holders. "Arm" is used to represent Arm Holdings plc; its operating company Arm Limited; and the regional subsidiaries Arm Inc.; Arm KK; Arm Korea Limited.; Arm Taiwan Limited; Arm France SAS; Arm Consulting (Shanghai) Co. Ltd.; Arm Germany GmbH; Arm Embedded Technologies Pvt. Ltd.; Arm Norway, AS, and Arm Sweden AB.

OpenCL OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Copyright