Logo, company name

Description automatically generated

**VMSS 2.0**

**New Generation of Aupera Video Machine Learning Streaming Server**

**Detailed User Guide**

**Document Revision: 2.0**

[1 Introduction to VMSS 2.0 4](#_Toc111029698)

[1.1 VMSS2.0 Pipelines 7](#_Toc111029699)

[2 Running Vmss2.0 on VMaccel 12](#_Toc111029700)

[2.1 Launching Aupera VMSS2.0 instance on VCK5000 12](#_Toc111029701)

[2.2 Using VMSS2.0 Web Client 13](#_Toc111029702)

[2.2.1 Running Aupera’s Crowd Flow Application 13](#_Toc111029703)

[2.2.2 Running Custom Pipelines 21](#_Toc111029704)

[2.3 Using VMSS2.0 Server (via command line) 28](#_Toc111029705)

[2.3.1 VMSS2.0 Server Docker 28](#_Toc111029706)

[2.4 Launching Your Own RTSP Streams 31](#_Toc111029707)

[3 Running VMss2.0 On-premiseS 32](#_Toc111029708)

[3.1 Prerequisites 32](#_Toc111029709)

[3.2 VMSS2.0 server (AVAS) 32](#_Toc111029710)

[3.3 Setting up RTSP Streams 33](#_Toc111029711)

[4 vmss2.0 pipelines 36](#_Toc111029712)

[4.1 Running VMSS2.0 Pipelines 36](#_Toc111029713)

[4.2 Pipeline Examples 36](#_Toc111029714)

[5 Aupera Node Toolkit 44](#_Toc111029715)

[5.1 Graph-level Configurations 44](#_Toc111029716)

[5.2 Node-level Configurations 45](#_Toc111029717)

[5.3 Nodes Currently Included in Aupera Node Toolkit 47](#_Toc111029718)

[5.3.1 Stream Demux 47](#_Toc111029719)

[5.3.2 Video Decoder 48](#_Toc111029720)

[5.3.3 Stream Mux 48](#_Toc111029721)

[5.3.4 Video Encode 49](#_Toc111029722)

[5.3.5 Video Filter 50](#_Toc111029723)

[5.3.6 Object (box) Detector 51](#_Toc111029724)

[5.3.7 Object Tracker 56](#_Toc111029725)

[5.3.8 Image Classifier 59](#_Toc111029726)

[5.3.9 Object (box) Visualizer 62](#_Toc111029727)

[5.3.10 Image Stream calculator 65](#_Toc111029728)

[5.3.11 Video Stream calculator 66](#_Toc111029729)

# Introduction to VMSS 2.0

Video Machine-learning Streaming Server (VMSS) is a software application designed to function as a ‘server’ process to provide video analytic services to multiple video streams and efficiently utilize multiple FPGA resources on a server system.

VMSS 1.0 was released in early 2021(VMSS 1.X) that had certain limitations. The limitations prevented the user from building more flexible ML pipelines. For instance, how plugins pass data to other nodes, network branching, packet synchronization across branches of ML pipelines, and in-order processing of frames across several streams that share the same network.

The new generation of VMSS (VMSS 2.0) has eliminated these limitations by replacing plugins with graph nodes that can have arbitrary types (as opposed to just pre-processing, post-processing, ML, and database). These nodes support back edges (edges from downstream nodes to the upstream ones) and side packets (less frequent communication, available during initialization, before any packets are received); thus, allowing arbitrary (and even cyclical) graphs instead of just the linear graphs that the previous generation supports. In addition, users can now choose to not use a single job queue for streams that share the same network; thus, allowing for packets to be synchronized across multiple streams.

Most importantly, with VMSS 2.0 pipelines can be built, configured, and run using a graphical user-interface allowing customers to rapidly create new applications using both the collection of nodes provided in Aupera’s node toolkit; or by creating their own nodes with the aid of Aupera’s node creation framework. When using Aupera’s node toolkit, building most ML pipelines does not require any coding (or hardware knowledge). Furthermore, using the graphical user interface customers can try any of the models on Xilinx Model Zoo (currently box detector and classifier models) with only a few mouse clicks.

VMSS 2.0 also provides a certain level of hardware abstraction (due to the higher modularity of nodes) which allows the same ML pipeline to run on different hardware platforms. This also allows for new hardware platforms to be adopted very quickly. Currently, pipelines built with VMSS 2.0 can run on VCK5000 acceleration cards (in addition to U50, U30, and AUPV205). In this release, we provide the VMSS 2.0 server along with a web-application that acts as the client (and visual GUI).

If you have additional questions and/or want to report an issue with your user experience, write to [vmss@auperatech.com](mailto:vmss@auperatech.com).

|  |  |  |  |
| --- | --- | --- | --- |
| Vendor | Modified | Size | Container Version |
| Aupera | TBD | 34.7GB | VMSS2.0\_AVAF2.0.4\_AVAS2.0.3\_VCK5000-prod |
| Aupera | TBD | 1.43GB | vmss2.0.0\_avac1.0.0 |

Shape

Deployment Options

This application is containerized and can be easily run in a few minutes on VCK5000 cards on an on-premises servers or on pre-configured cloud instances.

|  |  |
| --- | --- |
| **On Premises** | |
| VCK5000 | * Xilinx Runtime: **202120.2.12.427** * Target Platform: **xilinx\_vck5000\_gen3x16\_xdma\_base\_1** |

VMSS 2.0 consists of two major modules, Aupera Video AI Client (AVAC) and Aupera Video AI Server (AVAS). AVAC allows users to use a user-friendly GUI to connect to AVAS. Additionally, more advanced users may work with AVAS via the command line directly.

**NOTE:** We may use Aupera Video AI Client (AVAC) and Aupera’s VMSS2.0 AI Client interchangeably throughout this document. We may also use Aupera Video AI Server (AVAS) and Aupera’s VMSS2.0 AI Server interchangeably throughout this document.

The rest of this document is organized as follows:

Section 1.1 provides a brief introduction to VMSS2.0 pipelines.

Section 2.1 describes how to launch VMAccel instances that have both Aupera VMSS2.0 server and web client running.

Section 2.2 describes how to use the Aupera VMSS2.0 Web Client.

Section 2.3 describes how to use the Aupera VMSS2.0 Server via the command line.

Section 2.4 describes how to launch your own RTSP stream (optional).

Section 3.1 describes the prerequisites for installing and using VMSS2.0 on-premises.

Section 3.2 describes the process for installing VMS2.0 on-premises.

Section 3.3 describes how to set up the RTSP streaming services (both server and streams) on-premises.

Section 4.1 describes how to run VMSS2.0 pipelines using command line (both for on-premises setup and when using the VMAccel command line).

Section 4.2 describes the details of example pipelines.

Section 5.1 describes the configurations that are possible with Aupera Node Toolkit at the graph (i.e., pipeline) level.

Section 5.2 describes the configurations that are possible with Aupera Node Toolkit at the node level.

Section 5.3 lists the nodes that are currently available in Aupera Node Toolkit.

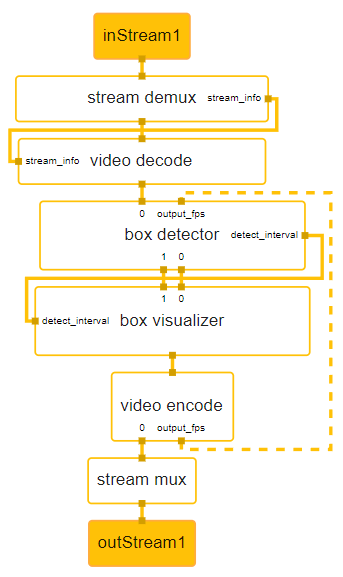
**NOTE:**

If you are using VMAccel, then you should read section 2. If you’re interested in using the command line (on VMAccel) or understanding more details about the provided pipeline examples, you can look at section 4.

If you are interested in an on-premises setup (i.e., setting up VMSS2.0 on your own machine hosting one or more VCK5000 cards), you need to read section 3. Afterwards, you can consult section 4 to learn how to run the provided pipeline examples.

## VMSS2.0 Pipelines

With VMSS 2.0, the entire pipeline is contained in a single proto (.pbtxt) file. This includes the names and parameters of all the nodes and their connections. You can visualize the pipeline’s pbtxt file using Aupera’s visualization tool (contained in AVAC when launching a custom pipeline) or any other third-party tool. In Figure 1.1, you can see a simple pipeline that showcases the major improvements added in VMSS 2.0.

  
**Figure 1.1. Aupera VMSS 2.0 major improvements pipeline**

The goal of above pipeline is to run an object detector on an input RTSP stream, visualize the detected bounding boxes on the frames, and return the results in an output RTSP stream. In most applications, latency is important; therefore, we have designed several mechanisms for improving overall pipeline latency (and throughput). This problem could be solved using a tracker but, in this example, a back-edge is used to send the output fps of the video encoder to the box detector. This way, the box detector can notice whether the output fps of the video encoder has fallen (or there is a backlog of frames to be encoded) and can automatically increase the detection interval (i.e., how often the box detector is run). You can also see that the detection interval of the box detector is communicated to the box visualizer node in real-time.

Below, you can see the content of the pbtxt file that describes the pipeline shown in Figure 1.1. As you can see, the box detector node will take the type of the model to run (ex. YoloV3, SSD, Refinedet, etc.) and the name of the specific kernel as input parameters. As such, this node can run any box detector model. To run your own model, all you need to do is place the compiled model along with its runtime prototxt config file (for AMD/Xilinx Model Zoo models, you can find this file provided along with the xmodel, please refer to an example provided [here](https://www.xilinx.com/bin/public/openDownload?filename=yolov3_voc-zcu102_zcu104_kv260-r2.5.0.tar.gz)) under the directory  **/usr/share/vitis\_ai\_library/models** in a folder matching the kernel name inside the AVAS docker. Detailed information about how to use the AVAS will be introduced later.

control\_port: 51881  
input\_stream: "inStream1"  
output\_stream: "outStream1"  
  
node {  
  name: "stream demux"  
  calculator: "stream\_demux"  
  input\_stream: "inStream1"  
  output\_stream: "packetstream1"  
  side\_node\_name: "decode"  
  side\_node\_name: "crowd\_flow"  
  output\_side\_packet: "stream\_info:stream\_info"  
  node\_options: {  
    [type.googleapis.com/gvis.StreamMuxOptions]: {  
      demux: {  
        rtsp\_transport: "tcp"    
        iframe\_extract: false  
        auto\_reconnect: true  
      }  
    }  
  }  
}  
  
node {  
  name: "video decode"  
  calculator: "x86\_dec"  
  input\_stream: "packetstream1"  
  output\_stream: "imgStream1080p"  
  side\_node\_name: "demux"  
  input\_side\_packet: "stream\_info:stream\_info"  
  node\_options: {  
    [type.googleapis.com/gvis.VideoCodecOptions]: {  
      dec: {  
        name: "x86\_dec\_h2645"    
        ow: 1920  
        oh: 1080  
        opixfmt: "BGR24"  
        queue\_size: 10000  
        low\_latency: false  
      }  
    }  
  }  
}  
  
node {  
  name: "box detector"  
  calculator: "box\_detector"  
  input\_stream: "imgStream1080p"  
  output\_stream: "detectionsStream"  
  output\_stream: "imgStream1080p\_detOut"  
  side\_node\_name: "tracker"  
  side\_node\_name: "crowd\_flow"  
  output\_side\_packet: "detect\_interval:detect\_interval"  
  stream\_sync\_mode: 1  
  stream\_sync\_maxwait\_ms: 60  
  input\_stream: "output\_fps:output\_fps"  
  input\_stream\_info: {  
    tag\_index: "output\_fps"  
    back\_edge: true  
  }  
  node\_options: {  
    [type.googleapis.com/gvis.BoxDetectorOptions]: {  
      detect\_interval: 5  
      detector\_type: "SSD"  
      kernel\_name: "RESNET18SSD\_ITER90000\_PRIVATE\_FINAL501IMAGES\_ADDED\_07JUNE2021\_CROWD\_FLOW\_PERSON\_HEAD"  
      need\_preprocess: true  
      log\_performance: true  
      run\_on\_letterboxed\_img: false  
      label\_confidence: {  
        label: 2  
        confidence: 0.5  
      }  
      inter\_class\_nms: {  
        labels: 2  
        threshold: 0.5  
      }  
    }  
  }  
}  
  
node {  
  name: "box visualizer"  
  calculator: "box\_visualizer"  
  input\_stream: "detectionsStream"  
  input\_stream: "imgStream1080p\_detOut"  
  output\_stream: "imgStream1080p\_aplOut"  
  side\_node\_name: "detector"  
  input\_side\_packet: "detect\_interval:detect\_interval"  
  stream\_sync\_mode: 1  
  stream\_sync\_maxwait\_ms: 60  
}  
  
node {  
  name: "video encode"  
  calculator: "x86\_enc"  
  input\_stream: "imgStream1080p\_aplOut"  
  output\_stream: "packetStream2"  
  output\_stream: "output\_fps:output\_fps"  
  node\_options: {  
    [type.googleapis.com/gvis.VideoCodecOptions]: {  
     enc: {  
       name: "x86\_enc\_h264"  
       w: 0   
       h: 0  
       fps: 0  
     }  
    }  
  }  
  side\_node\_name: "vfilter\_node"  
  side\_node\_name: "mux\_node"  
}  
  
node {  
  name: "stream mux"  
  calculator: "stream\_mux"  
  input\_stream: "packetStream2"  
  output\_stream: "outStream1"  
  node\_options: {  
    [type.googleapis.com/gvis.StreamMuxOptions]: {  
     mux: {  
       rtsp\_transport: "tcp"  
       auto\_reconnect: true  
     }  
    }  
  }  
  side\_node\_name: "encode\_node"  
}

# Running Vmss2.0 on VMaccel

In this section, we will introduce the steps to sign-up for the VMAccel demo account, access the Aupera VMSS2.0 instance, launch custom RTSP streams for tasks, and run Aupera VMSS2.0 pipelines through both the Aupera Video AI Client (AVAC) and Aupera Video AI Server (AVAS) with detailed examples, including crowd flow pipeline and custom pipelines.

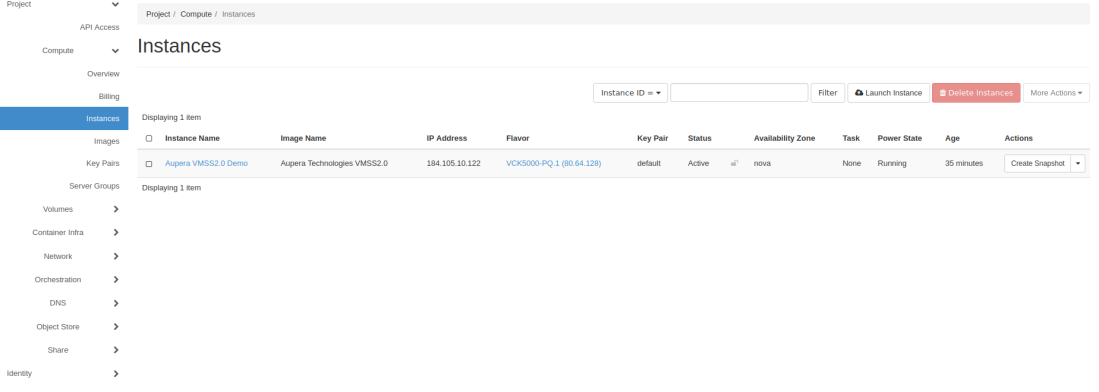
## 2.1 Launching Aupera VMSS2.0 instance on VCK5000

Please sign up for a demo account at: <https://www.vmaccel.com/vmssdemo>

After completing the sign-up form, you will receive an email with your demo credentials. Please follow the instructions in the email to log into VMAccel.

**NOTE:** The trial accounts allow 5 hours of free evaluation of VMSS2.0. The trial accounts are currently configured to automatically delete any instances when a user logs out. You may use a total of 5 hours of runtime, and this could be extended over several days. Your account will be locked once you have depleted 5 hours of runtime.

For each user, a VMAccel instance will be automatically launched. Once you login, under *Project-> Compute* you should see your instance by selecting “*Instances*” in the left-hand sidebar. You should be able to see the instance that was just created for you being spawned as shown in Figure 2.1.



**Figure 2.1. VMAccel instances main page**

After some time, the status of the instance will change to Active. At this point, the installation of the VMSS2.0 server and the client on the instance begin. Please allow about 3 minutes for this process to finish after you see the status has changed to Active.

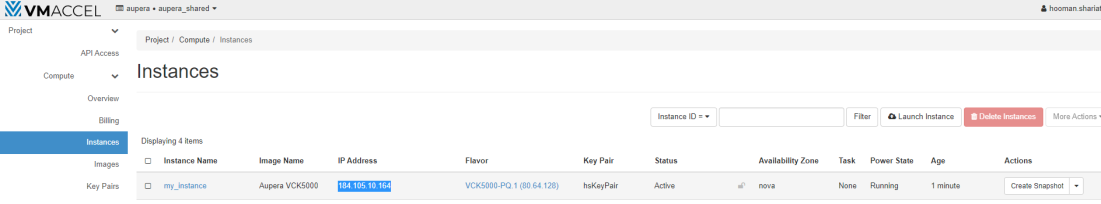
**NOTE 1:** To run VMSS2.0 pipelines, you can either use Aupera’s VMSS2.0 Web Client (as described in section 2.2) or access the VMSS.2.0 server (as described in section 2.3) using the command line.

**NOTE 2:** To facilitate testing, the VMAccel instance will automatically start two RTSP streams **rtsp://<vmaccel\_instance\_ip\_address>:8554/stream1** and **rtsp://<vmaccel\_instance\_ip\_address>:8554/stream2**. **vmaccel\_instance\_ip\_address** is the IP address that is shown in the image above under IP Address when you are in *Project->Compute->Instances*. The first stream contains a crowded scene of people passing by and is the most useful for testing head, person, and other human related detections. We use this video for benchmarking our crowd applications. The second stream contains objects usually encountered in a retail scenario. We use this stream for throughput benchmarking.

Additionally, AVAC, the VMSS2.0 Web Client, can connect to any RTSP streams that are broadcast on open ports.

## 2.2 Using VMSS2.0 Web Client

After launching an Aupera VMSS2.0 instance on VMAccel, you can access the web client by copying the IP address of your instance into your browser on your local machine; and adding the port 3004. For example, in the screenshot below, the IP address of the instance is 184.105.10.164 which means that the web client can be accessed by typing <http://184.105.10.164:3004/> in your browser.

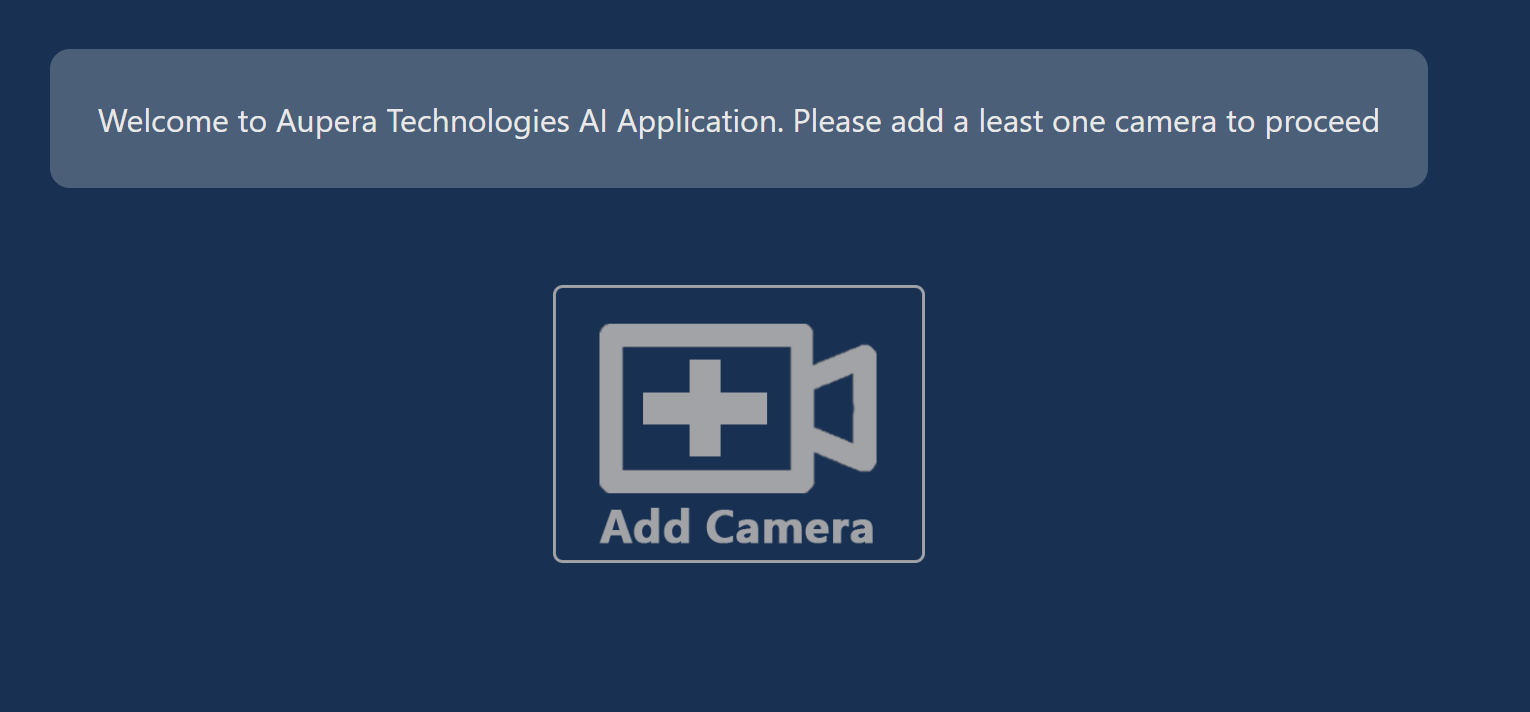


**Figure 2.2. VMAccel instances page with instance IP address highlighted**

**NOTE:** Since the relevant ports of your VMAccel instance are open to the public, you do not need to use the browser of the VMAccel instance (accessible through VNC) to launch the Web Client. You can use the browser on any machine with an internet connection.

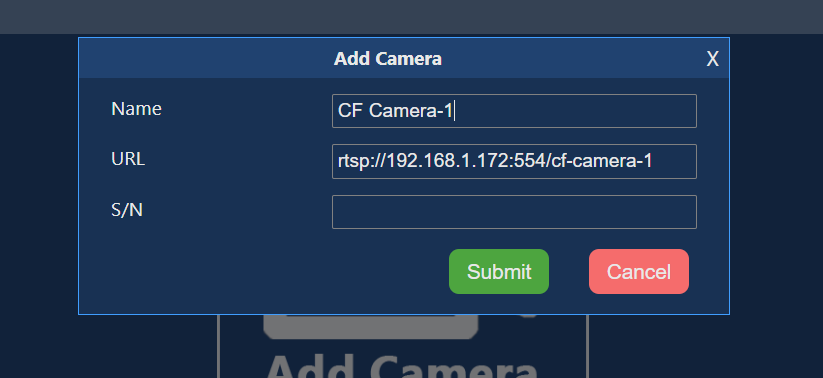
### 2.2.1 Running Aupera’s Crowd Flow Application

A. Click ***Add Camera*** to add at least one camera to access the Application functions.



**Figure 2.3. Aupera web application page**

B. Enter a ***Name*** (any arbitrary name can be used)and ***URL*** (***S/N***is not required). **Make sure that RTSP URL is correct.** You can right-click on the added camera to access the camera configuration shown in Figure 2.4.

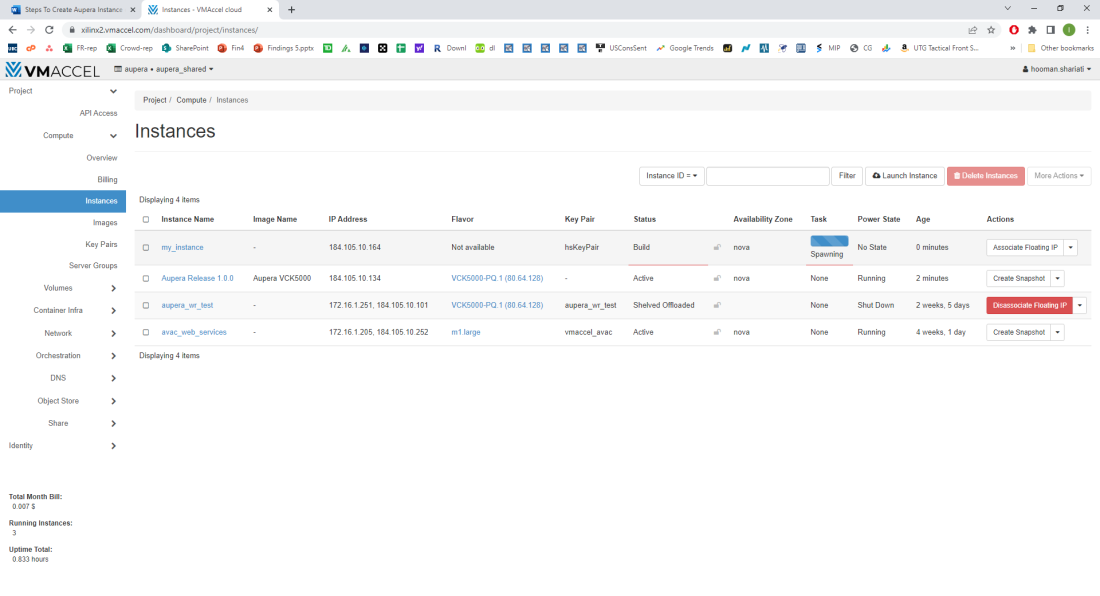


**Figure 2.4. Aupera web application page – add camera**

**NOTE:** You can use either your own publicly available RTSP stream (perhaps the one you launched using the instructions in Section 2.4); or it can be one of the two RTSP streams that automatically started up when you launched your VMAccel instance.

For the latter you can use **rtsp://<vmaccel\_instance\_ip\_address>:8554/stream1** and **rtsp://<vmaccel\_instance\_ip\_address>:8554/stream2**.

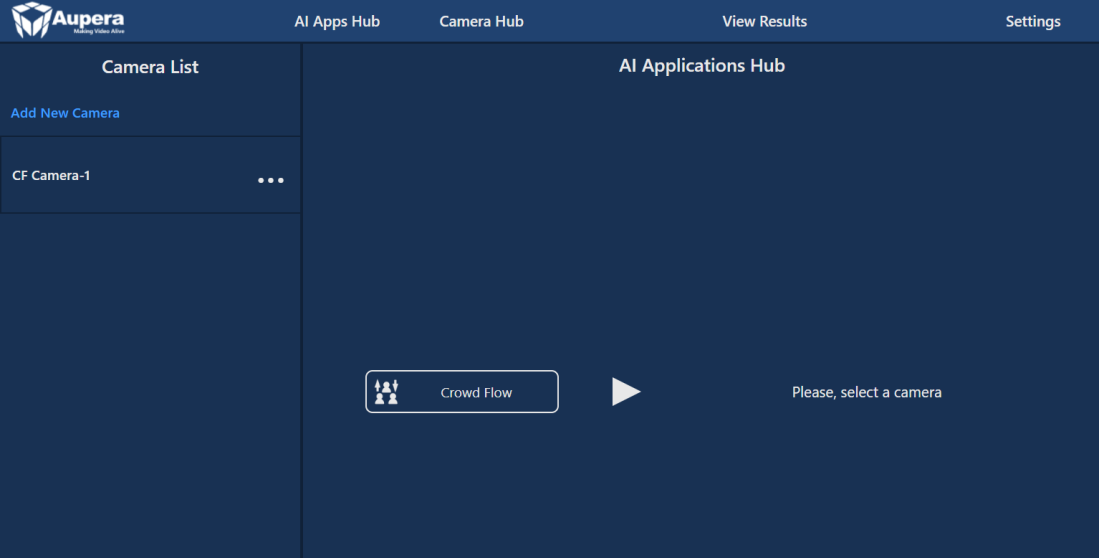
Just make sure to replace the **vmaccel\_instance\_ip\_address** with the IP address of the VMAccel instance you just launched. You can find this IP address by clicking on the left-hand sidebar select “Instances” you can see the instance you just created being spawned as shown in Figure 2.5.



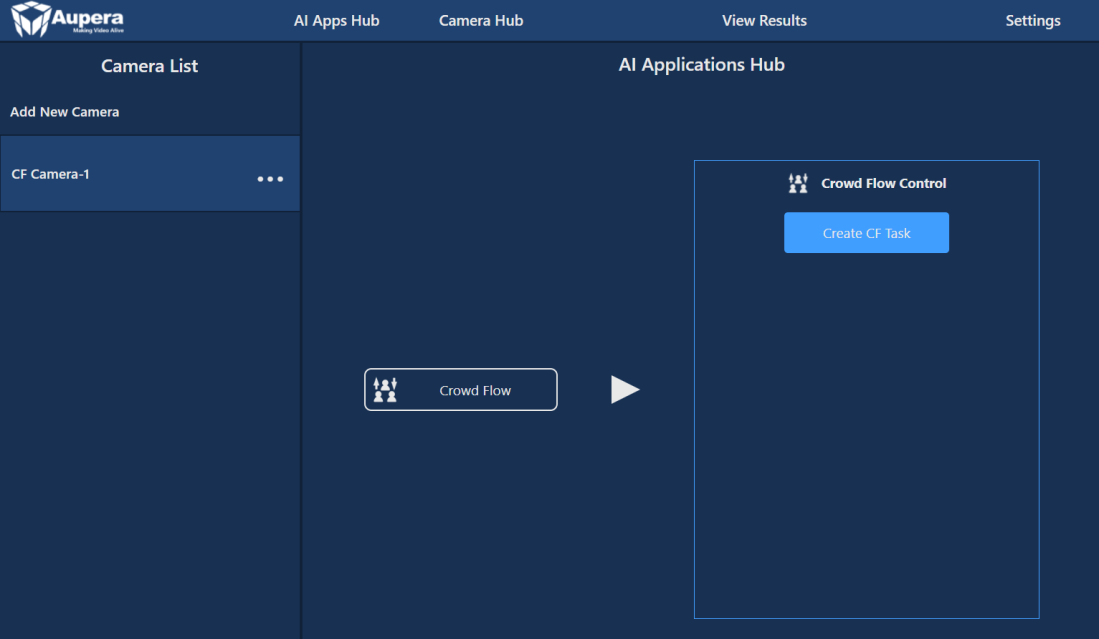
**Figure 2.5. VMAccel instances page – preparing a new instance**

Again, the first stream shows a scene with people (used for crowd, person, and person attributed applications) while the second stream shows a scene with retail objects.

C. Click ***AI Apps Hub***, then click on a camera in the ***Camera List***, after that the Crowd Flow Control will appear.

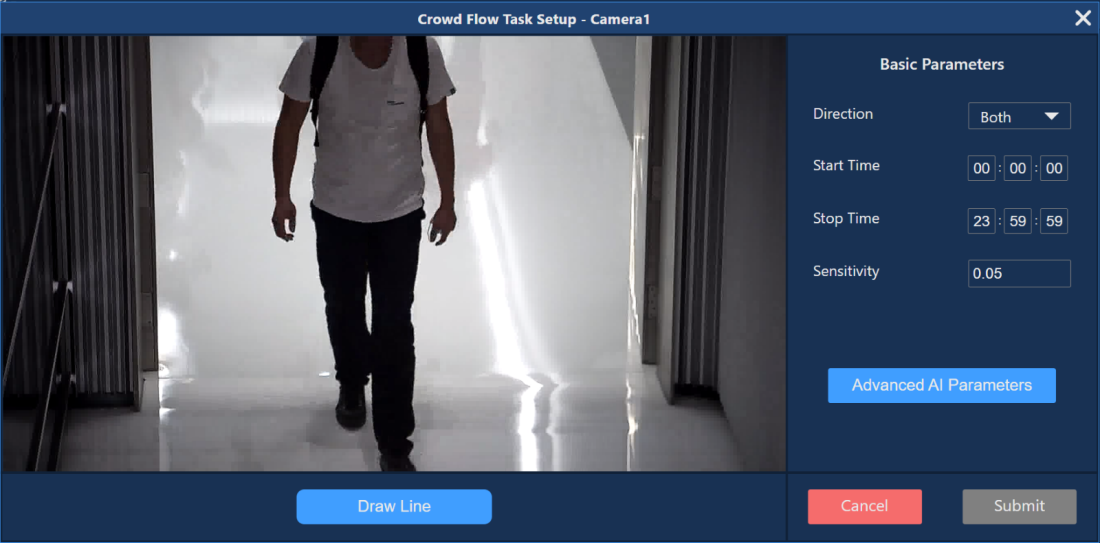


**Figure 2.6. Aupera web application page – crowd flow task main page**



**Figure 2.7. Aupera web application page – crowd flow task linking to camera**

D. Click ***Create CF Task*** in Crowd Flow Control, after that the Crowd Flow Task Setup window will appear.



**Figure 2.8. Aupera web application page – crowd flow task creation**

E. You need to draw line(s) to indicate a border for people crossing. When a person’s head crosses the line, IN/OUT count will reflect this event. These border lines may consist of up to a maximum of 14 segments.

1. To start drawing, click the ***Draw Line*** button, the cursor will change to a cross.

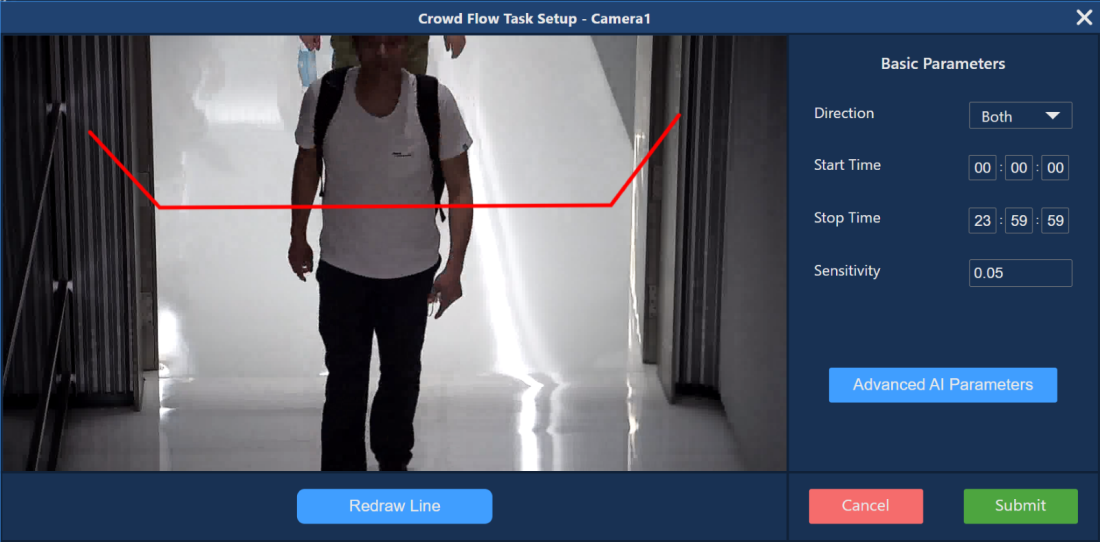
2. **Left-click** and **hold the mouse button** on the place where you would like to start the line.

3. Drag the Line to the place where you want to finish the first segment, then release the mouse button.

4. Move a mouse to the end of the next segment to complete it.

5. To finish drawing, click the **Right mouse button**, unfinished segment will be deleted.

After the Line is drawn, “Draw Line” will change to “Redraw Line”. Click it if you want to delete the Line drawn and start drawing from the beginning. It is recommended to draw U shaped lines as shown in Figure 2.9 to capture people who may move parallel to the line and around it.



**Figure 2.9. Aupera web application page – crowd flow task lines drawing example**

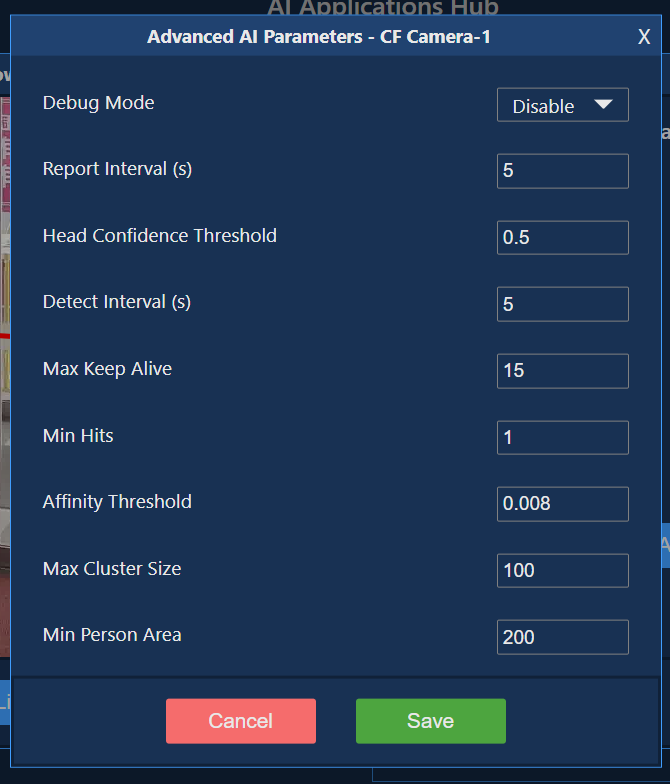
F. Changing Basic Parameters is not required to start the task, you can keep default values.

1. ***Direction*** – Count people going “In”, “Out” (“Entering” / “Exiting”) or both directions

2. ***Start/Stop Time*** – counting will be started and stopped at the given time every day;

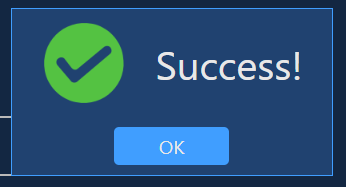
3. It is recommended to set ***Sensitivity*** to the default value of 0.05;

G. Advanced Parameters can significantly affect the results for a particular task, it is recommended to not change those until recommended by Aupera.



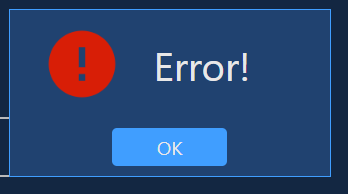
**Figure 2.10. Aupera web application page – crowd flow task advanced AI parameters**

H. To start the task, click the ***Submit*** button. After that, a pop-up message will notify you that the task was successfully launched.



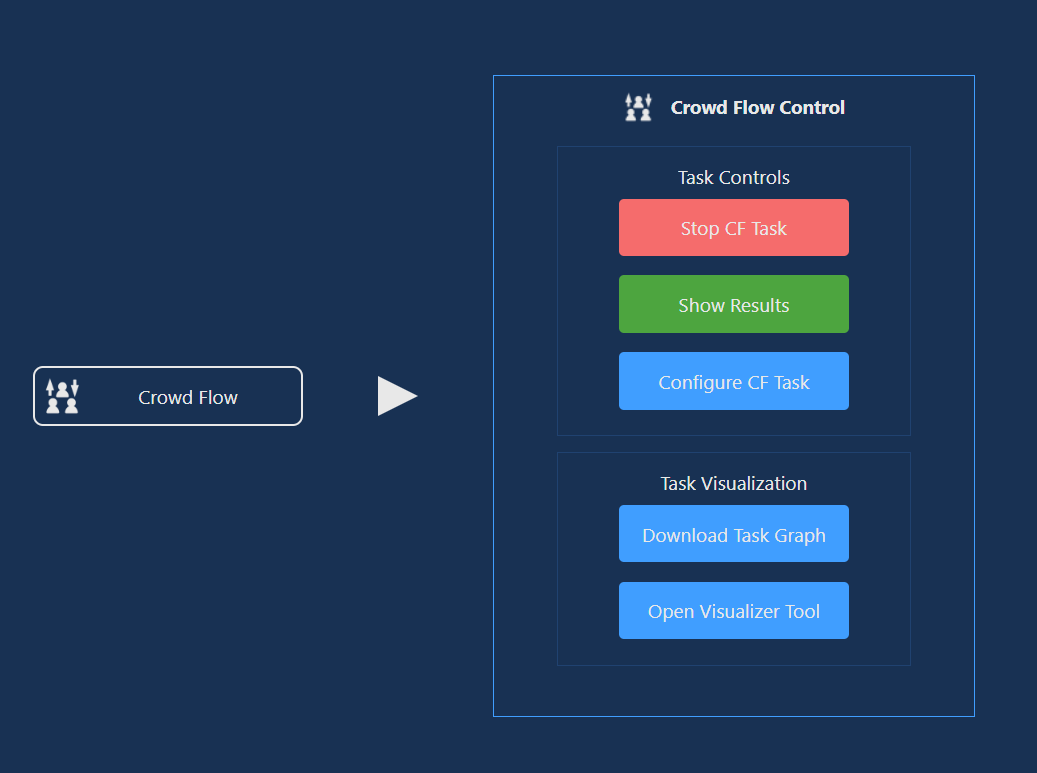
**Figure 2.11. Aupera web application page – success notification**

I. If the pop-up message reports an error, try launching the task with default parameters or check the settings.



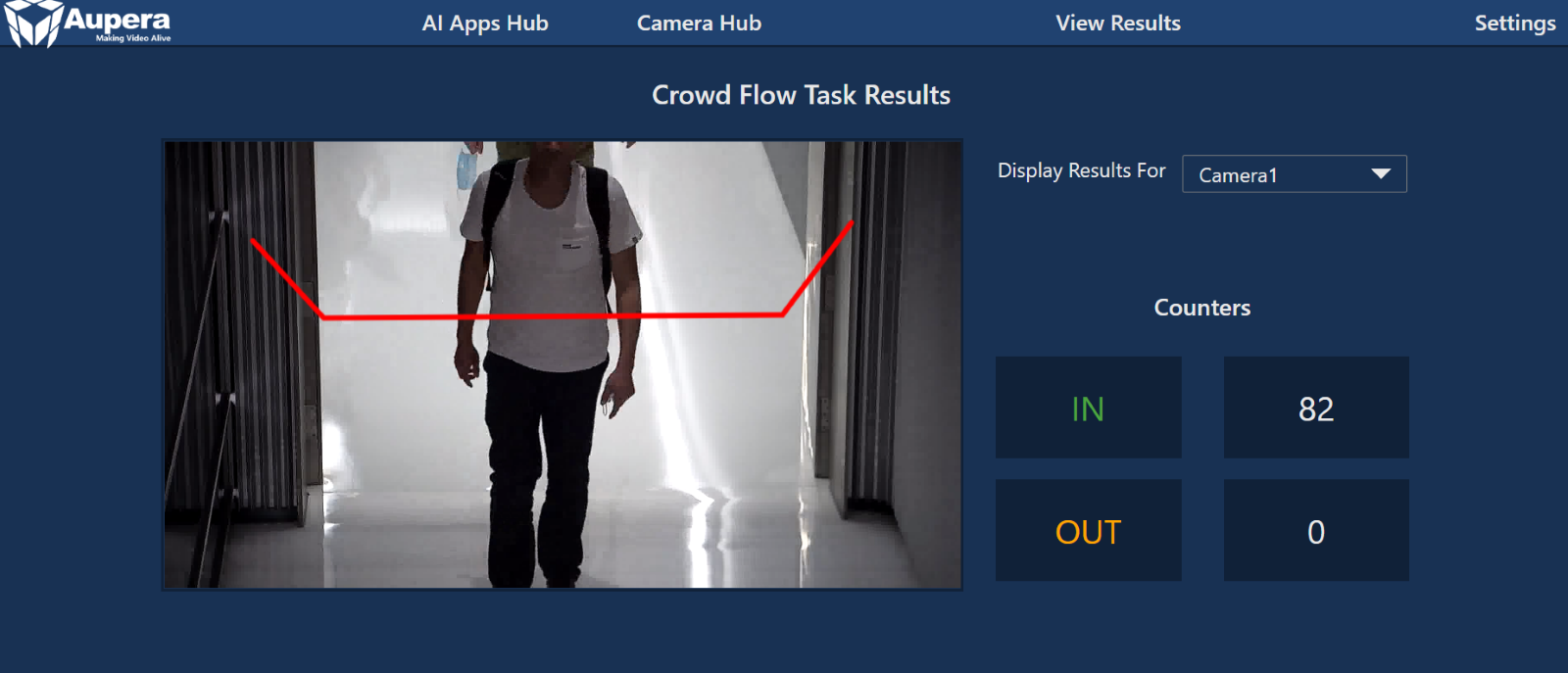
**Figure 2.12. Aupera web application page – error notification**

J. If the task was launched, the Crowd Flow Control will change its view and reveal additional buttons.



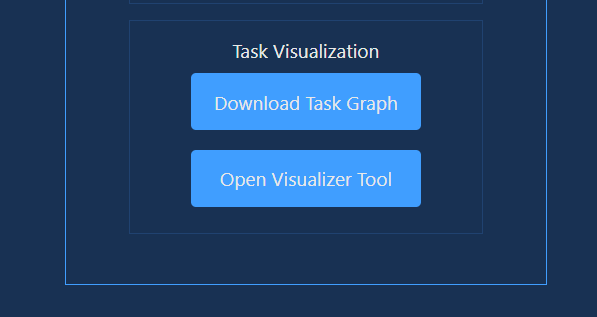
**Figure 2.13. Aupera web application page – crowd flow task control**

K. To view results, click the ***Show Results*** button. You will be redirected to the Results page and a corresponding camera will be selected automatically. If you have more than one task launched, you can switch between them with the ***Display Results For*** drop-down box.



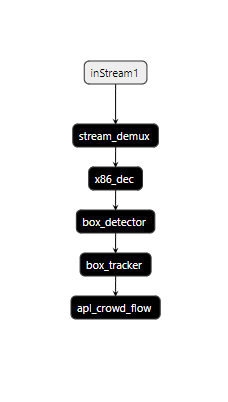
**Figure 2.14. Aupera web application page – crowd flow task result**

L. To visualize the AI pipeline for the current task, under AI Apps Hub -> CF Task -> click the ***Open Visualizer Tool Button****.*



**Figure 2.15. Aupera web application page – crowd flow task visualizer tool**

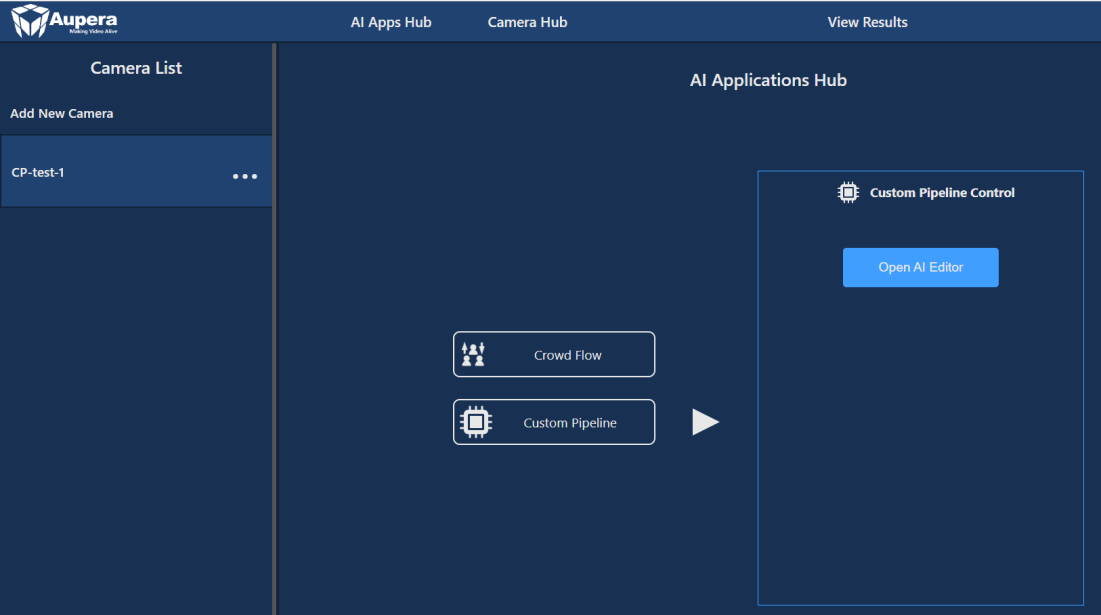
**NOTE:** You cannot download the pipeline graph (.pbtxt file) for Crowd Flow and run it as a custom pipeline. This is because the input and output RTSP streams are not configured properly in the pipeline graph (.pbtxt file) that you will be downloading. To run custom pipelines please use the examples explained in section 4.2 (accessible [here](https://auperatechvancouver.sharepoint.com/:f:/g/EhT4JMyXkWJOr0MQBp9y5GUBgoAnKUYJ5R2dpgmROIKMtw?e=CnfzTQ)). If you’re running Custom Pipeline through Aupera Web Client, please only use the files that include “using\_rtsp” in their name.

M. The Visualizer will open in a new tab. Click ***Open Model...*** and select the file you downloaded on the previous step. Then, the AI pipeline graph will appear.

**Figure 2.16. Aupera web application page – crowd flow task AI pipeline graph**

### 2.2.2 Running Custom Pipelines

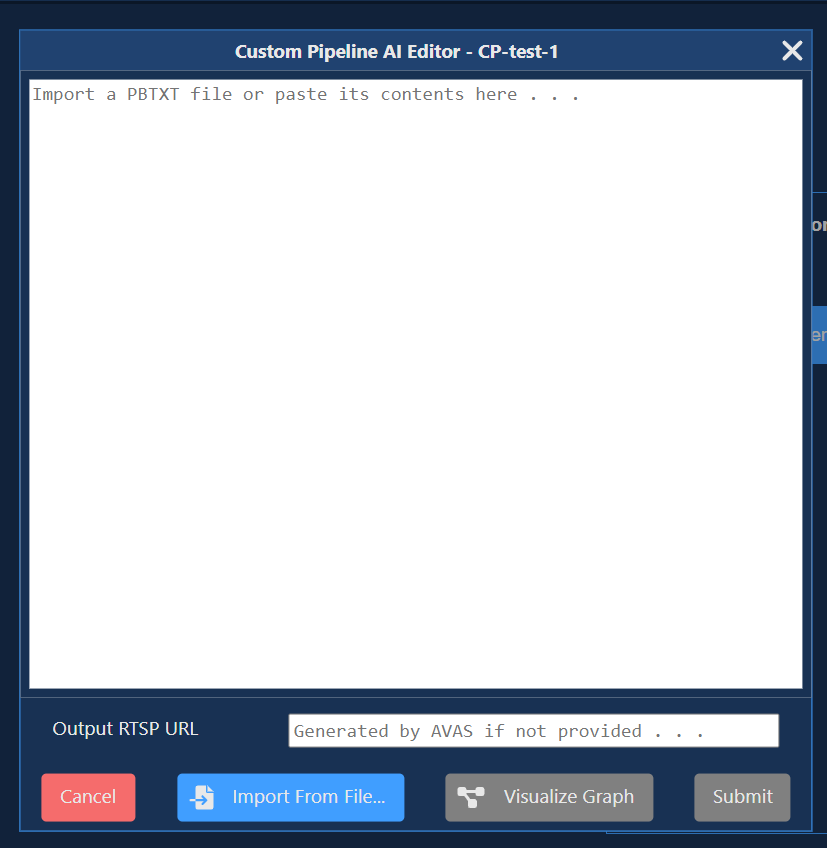
A. Click on a camera, then on the ***Custom Pipeline*** button, after that the CP control component will appear



**Figure 2.17. Aupera web application page – custom pipeline task main page**

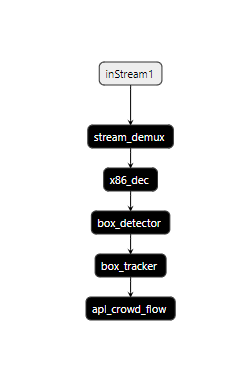
B. Click ***Open AI Editor***. In the Editor you can type a PBTEXT configuration or ***Import From File*** it from a file by clicking the corresponding button.

**NOTE:** To run custom pipelines, please use the examples explained in section 4.2 (accessible [here](https://auperatechvancouver.sharepoint.com/:f:/g/EhT4JMyXkWJOr0MQBp9y5GUBgoAnKUYJ5R2dpgmROIKMtw?e=CnfzTQ)). If you’re running Custom Pipeline through Aupera Web Client, please only use the files that include “using\_rtsp” in their name.



**Figure 2.18. Aupera web application page – custom pipeline task AI editor**

C. After PBTEXT has been typed or imported, it can be visualized. Clicking on the “*Visualize*” button will open a new tab in which the Graph will be displayed.

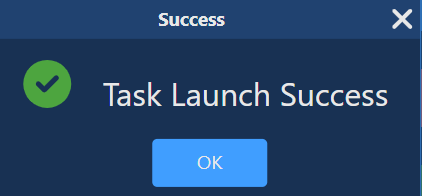


**Figure 2.19. Aupera web application page – custom pipeline task AI pipeline graph**

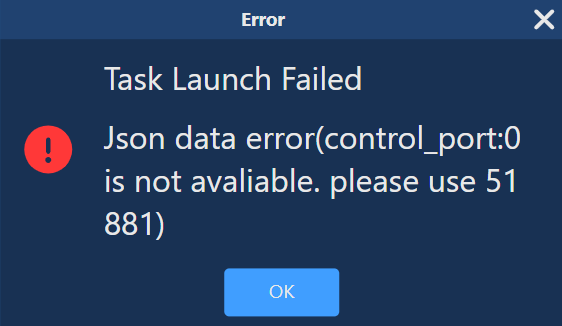
D. Before starting the task, please enter *Output RTSP URL* (this value needs to follow a specific format of **rtsp://<vmaccel\_instance\_ip\_address>:8554/<user\_specifcied \_name>**). Click Submit to start the CP task.

**NOTE:** **<user\_specifcied\_name>** can be any arbitrary name that the user chooses.

E. Whether the task was started successfully or not, a corresponding message will be displayed as the pop-up.

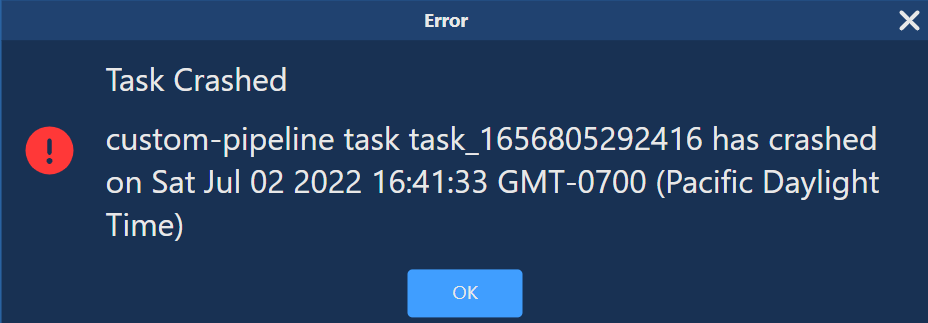


**Figure 2.20. Aupera web application page – task launch success notification**



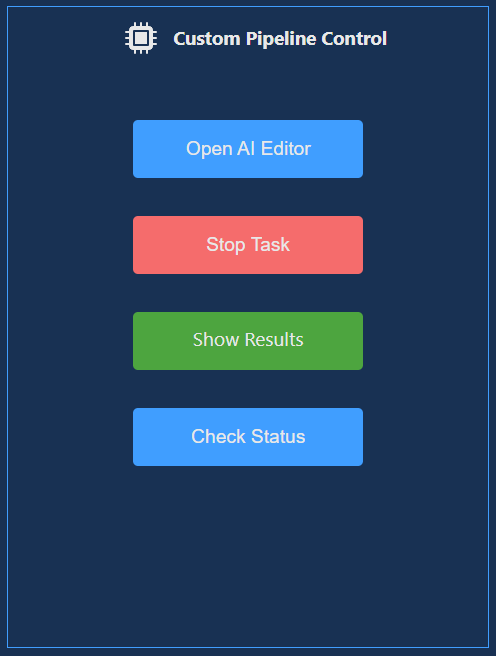
**Figure 2.21. Aupera web application page – task launch failed notification**

F. If the task was started, but then crashed after some time, message about that will be displayed



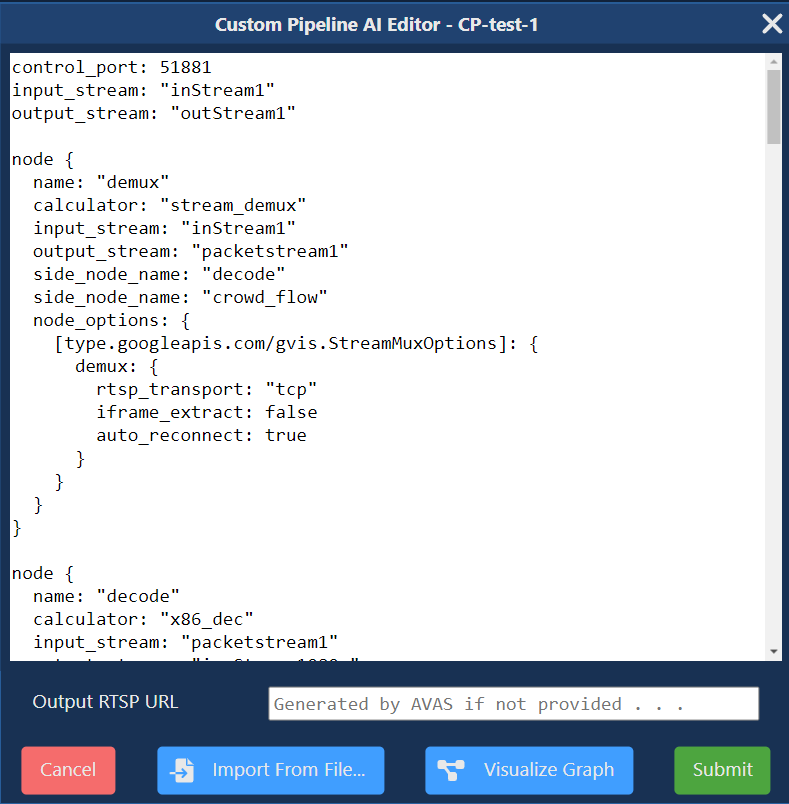
**Figure 2.22. Aupera web application page – task crashed notification**

G. When the task is launched, the CP control component will change its layout and will offer additional options.



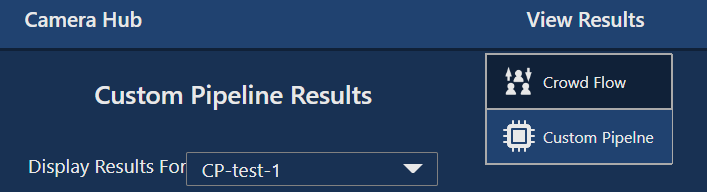
**Figure 2.23. Aupera web application page – custom pipeline task control**

H. If AI Editor is opened for a running task, current PBTEXT configuration will be displayed in the editor field. However, task update is not supported now, so please stop and start the task again in case any changes in PBTEXT are required.



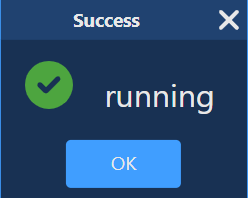
**Figure 2.24. Aupera web application page – custom pipeline task pbtxt example**

I. To see the CP task results (output video), either click on ***Show Results*** in the CP control component or navigate to CP Results using the Header, then choose desired camera in the ***Display Results For*** list.



**Figure 2.25. Aupera web application page – custom pipeline task result**

J. Current state of the task can be checked with the Check Status button.



**Figure 2.26. Aupera web application page – task success running notification**

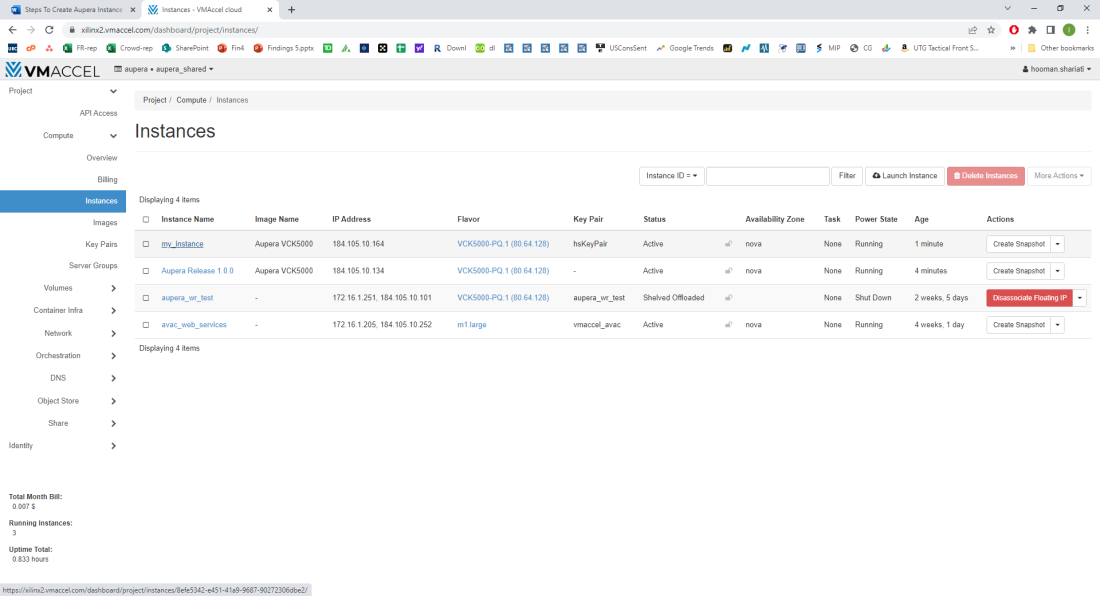
K. To stop the task, click Stop Task in the CP control component.

## 2.3 Using VMSS2.0 Server (via command line)

To use AVAS, the VMSS2.0 Server, you need to launch a VMAccel terminal and go into the docker for VMSS2.0 Server. Once there, you can run any pipelines directly from the command line.

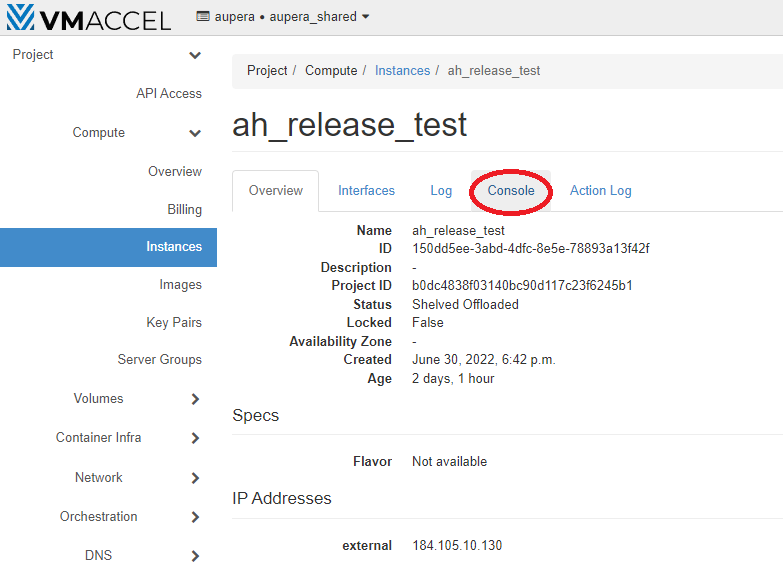
### 2.3.1 VMSS2.0 Server Docker

A. In the left-hand sidebar select **“*Instances*”.** Then, click on the name of your instance.



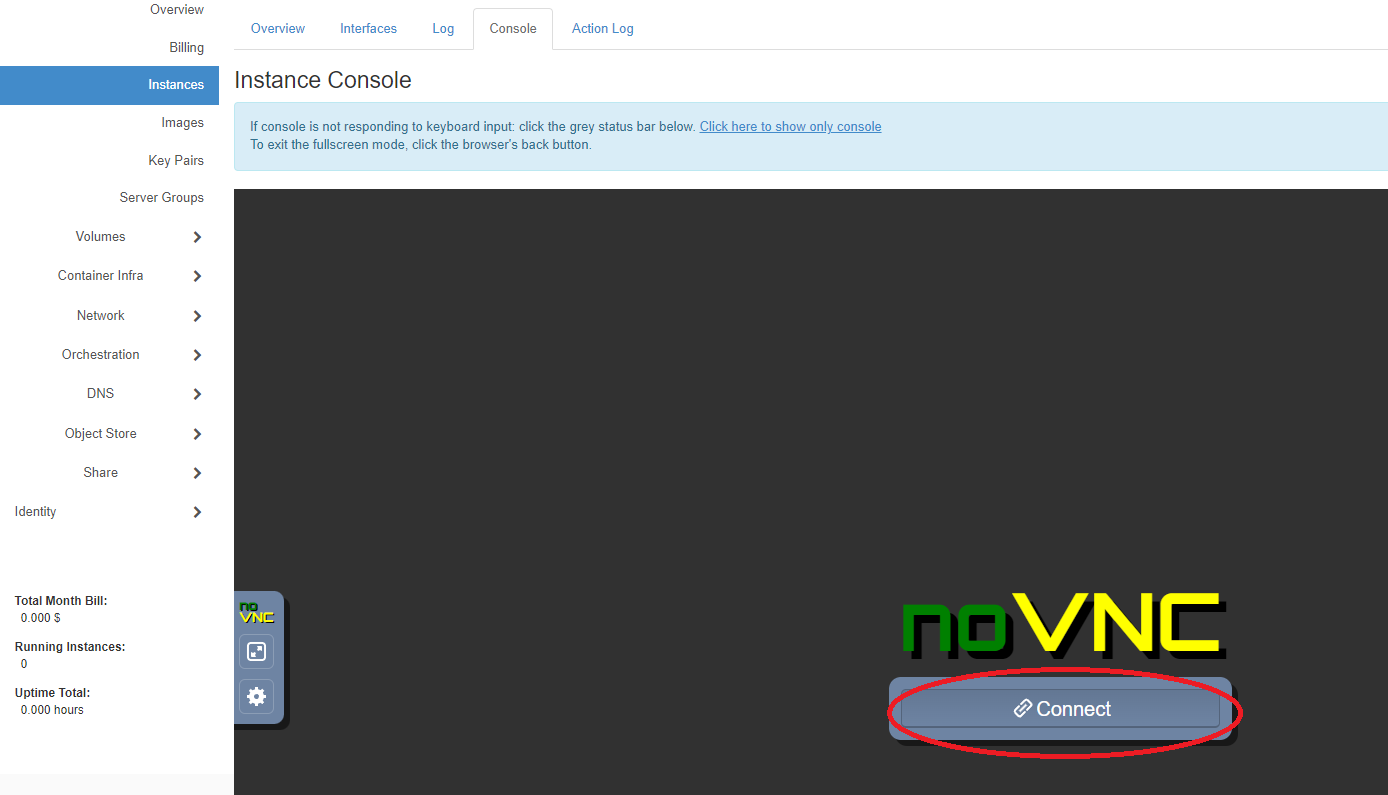
**Figure 2.27. VMAccel instances page – instance selection**

B. Select the ***Console*** tab as Figure 2.28 shows.



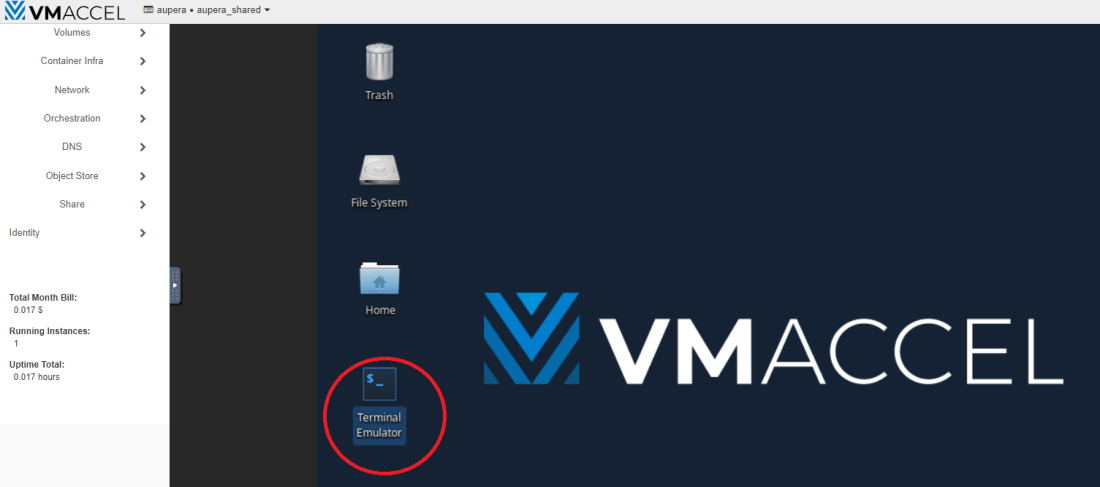
**Figure 2.28. VMAccel instances page – instance console**

C. Once the VNC window opens click on Connect as Figure 2.29 shows.



**Figure 2.29. VMAccel console page – instance console connection**

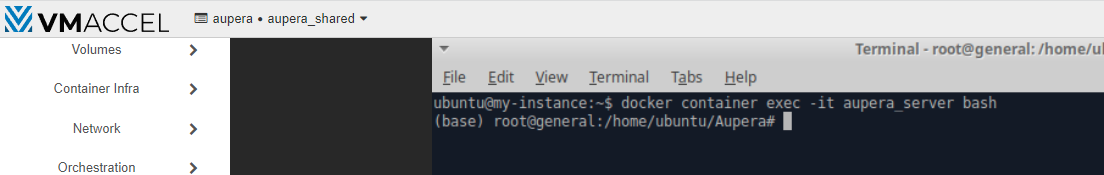
D. Inside the VMAccel VNC window, click on the Terminal icon to open a command line shell terminal.



**Figure 2.30. VMAccel console page – terminal emulator highlighted**

E. From the terminal, you can enter the VMSS2.0 server’s docker by running the command:

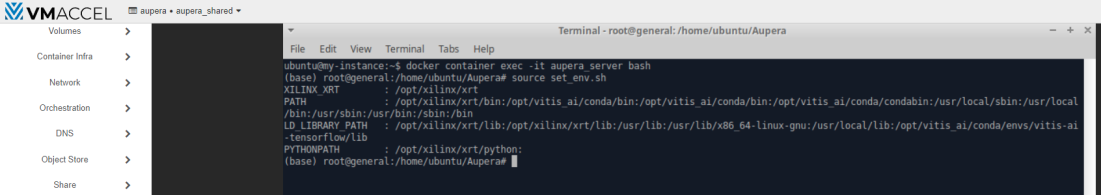
docker container exec -it aupera\_server bash



**Figure 2.31. VMAccel console page – accessing VMSS2.0 server’s docker**

F. Once inside the docker, you will need to setup the environment (xbutil and vitis) by running the command below from any directory.

source set\_env.sh



**Figure 2.32. VMAccel console page – setting up software environment**

At this point, the docker is ready to be used and you can proceed to section 4.

## 2.4 Launching Your Own RTSP Streams

You can run VMSS2.0 pipelines either on RTSP streams (including the one from your IP camera) or on videos. If you are using the RTSP streams provided (mentioned above under NOTE 2 in Section 2.1), you can skip this section and move on to section 2.3.

This section focuses on helping the user to broadcast their test videos via an RTSP server. Before continuing, please make sure FFMPEG is downloaded on your local machine and accessible via the command line. Visit the [FFMPEG download](https://ffmpeg.org/download.html) page for further instructions.

For your video file residing on your local machine to be pushed into a RTSP server, you can follow the instructions below:

1. You can publish a stream using

ffmpeg -re -stream\_loop -1 -i file.ts -c copy -f rtsp rtsp://server\_ip:8554/mystream

Where *file.ts* is your video file residing on your local machine and *server\_ip* is the IP address of your VMAccel instance.

**NOTE:** The key *mystream (*in *rtsp://server\_ip:8554/mystream* above) should be a unique string for each stream.

1. You can then watch the stream using VLC (or any other video player) through **media/open network** stream option, by hitting *ctrl+n* or by using

vlc rtsp://server\_ip:8554/mystream

You can download VLC from [here](https://www.videolan.org/).

# Running VMss2.0 On-premiseS

## Prerequisites

A. machine with X86 processor running Ubuntu 18.04 and containing at least one VCK5000 board (along with PCIe x16) to host the VMSS 2.0 server. From now on, this machine will be referred to as the X86 host

B. machine to host the VMSS 2.0 client (web-application). This machine could run Linux, or Windows. Optionally, it could, also, be the same machine hosting the server.

C. docker version 20.10.8 or above installed. You can check this using

sudo docker --version

D. docker compose version v2.4.1 or above installed. You can check this using

sudo docker compose version

## VMSS2.0 server (AVAS)

A. Load the docker using

sudo docker load -i VMSS2.0\_AVAFx.x.x\_AVASx.x.x\_VCK5000-prod.tar.gz

B. Download VMSS2.0 server docker (VMSS2.0\_AVAF2.0.0\_AVAS2.0.0\_VCK5000-prod) and the docker\_run\_2.sh script from OneDrive and upload them onto the X86 host (the machine hosting the VCK5000 card)

C. Start the VMSS 2.0 server docker using the provided docker\_run\_2.sh script. By running

sudo ./docker\_run\_2.sh auperastor/video\_ai\_framework:VMSS2.0\_AVAFx.x.x\_AVASx.x.x\_VCK5000-prod VMSS2.0\_AVAFx.x.x\_AVASx.x.x\_VCK5000-prod

You might need to change the permissions of docker\_run\_2.sh before running it using

sudo chmod 7777 docker\_run\_2.sh

D. Please note that our VMSS2.0 server dockers are locked using device serial numbers. In order to use our docker, you will need to send us the serial number of the device that is intended to run the server. In turn, we will provided you with a public-private key pair. You will then need to place both the public key (e.x. pubkey.pem) and the private key (e.x. key.json) in the **/opt/aupera/avas/etc/** folder of the docker before running the start.sh script.

E. If you’re planning on only using the web-application to run tasks, you can run the docker in detached mode using

sudo docker exec -it VMSS2.0\_AVAFx.x.x\_AVASx.x.x\_VCK5000-prod bash start.sh

F. To see the debugging screen you can use

sudo docker exec -it VMSS2.0\_AVAFx.x.x\_AVASx.x.x\_VCK5000-prod bash start.sh debug

You can exit the debugging screen with **Ctrl+A D** to exit debug screen

G. If you’re planning on using the command line server (instead of the web-application), instead of above two steps (steps 4 and 5), you can go inside the container using

sudo docker exec -ti container\_name bash

And then start the server (listening to the client) by running **start.sh** in any directory (from inside the docker)

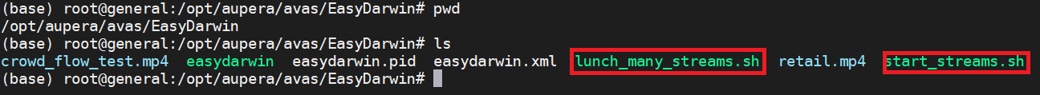
H. If you are not running the start.sh script (i.e., you’re planning on just using the command line to run pipelines without using the client/ avas server), then you need to source the xbutil and Vitis environments by running the command:

source set\_env.sh

The set\_env.sh script is on the path (located in **/opt/aupera/avas/etc/**) so it the command above can be ran from any locations inside the docker.

## Setting up RTSP Streams

Inside the VMSS2.0 server’s docker, you will find a folder called **/opt/aupera/avas/EasyDarwin** as shown in the screenshot below:



Inside of this (EasyDarwin) folder, you will find two scripts:

1. If you run the command: “./start\_strams.sh” you will start the server and two streams. stream1 is a crowd video and stream2 is a retail video.

* If you run this script (in a separate shell) you will be able to run all the examples except throughput\_benchmarking\_using\_retail\_application.
* If you’d like to use rtsp run:

avaser -i input.pbtxt -o output.pbtxt using\_rtsp...pbtxt

* If you’d like to use video run:

avaser -i input.pbtxt -o output.pbtxt using\_video.pbtxt

* using both rtsp and video. You can see the output rtsp at:

rtsp://yourMachineIP:554/out1

(just replace yourMachineIP with the actual IP address of your machine)

1. The second script lunches many rtsp streams. This is to test the throughput\_benchmarking\_using\_retail\_application example.
   * You can lunch the streams by running:

./lunch\_many\_streams.sh #ofStreams

* + For example, to test 37 streams you should run:

./lunch\_many\_streams.sh 37

* + To test 56 streams you should run:

./lunch\_many\_streams.sh 56

* + If you are running the 37 examples with video output using the command

avaser -i input.pbtxt -o output.pbtxt -c config\_withTracker\_withVideoOut.pbtxt

* + Then you can watch the output video streams at:

rtsp://yourMachineIP:554/out1” through “rtsp://yourMachineIP:554/out37

(just replace yourMachineIP with the actual IP address of your machine)

* + If you are running the 37 examples without video output, please make sure to use the empty.pbtxt as output (passed by -o options). Basically, the command you would run would be:

avaser -i input.pbtxt -o empty.pbtxt -c config\_noTracker\_noVideoOut.pbtxt

# vmss2.0 pipelines

## 4.1 Running VMSS2.0 Pipelines

Generally, to run a VMSS2.0 pipeline, you can run the command below from any directory inside of the VMSS 2.0 server docker. There are 3 pbtxt files that are required to pass to *avaser*:

1. **Input**: comes after *-i* parameter and contains the same number of RTSP streams as the input\_streams contained in your pipeline.pbtxt.
2. **Output**: comes after *-o* parameters and contains the same number of rtsp streams (or file passes) as the output\_streams contained in your pipeline.pbtxt.
3. **Config**: comes after *-c* parameter and contains your pipeline definition (the list of nodes and connections).

Below as an example of what the command should look like:

avaser -i input.pbtxt -o output.pbtxt -c pipeline.pbtxt

Below is an example of the content of an input.pbtxt file:

input\_urls: "rtsp://10.10.190.114:554/key1"  
input\_urls: "rtsp://10.10.190.114:554/key2"  
input\_urls: "rtsp://10.10.190.114:554/key3"

Below is an example of the content of an output.pbtxt file:

output\_urls: "rtsp://10.10.190.114:554/key4"  
output\_urls: "rtsp://10.10.190.114:554/key5"  
output\_urls: "/tmp/output\_video\_file.mp4"

**NOTE 1:** The output.pbtxt file could be empty if there are no output streams.

**NOTE 2:** The output.pbtxt file could contain file paths instead, in which case, the encoded video will be saved to disk instead of being sent to the RTSP streaming server.

## 4.2 Pipeline Examples

We have provided several examples of full pipelines [here.](https://auperatechvancouver.sharepoint.com/:f:/g/EhT4JMyXkWJOr0MQBp9y5GUBgoAnKUYJ5R2dpgmROIKMtw?e=CnfzTQ) These are also included in the aupera\_server docker in the **/opt/aupera/avas/examples** folder. You can navigate to this location using the following command:

cd /opt/aupera/avas/examples

In most of the example folders there are two sets of pipelines pbtxt files: one called **using\_rtsp\_...pbtxt** and another called **using\_video.pbtxt**.

If you’d like to try the example pipelines on the sample videos, then all you need to do is to go inside the sub-folder of a specific example (box\_detector, box\_detector\_classifier\_cascade, or apl\_crowd\_flow) and run the following command:

 avaser -i input.pbtxt -o output.pbtxt -c using\_video.pbtxt

If you’d like to try the example pipelines on the RTSP streams that are automatically started by your VMAccel instance, then all you need to do is go inside the sub-folder of a specific example and run:

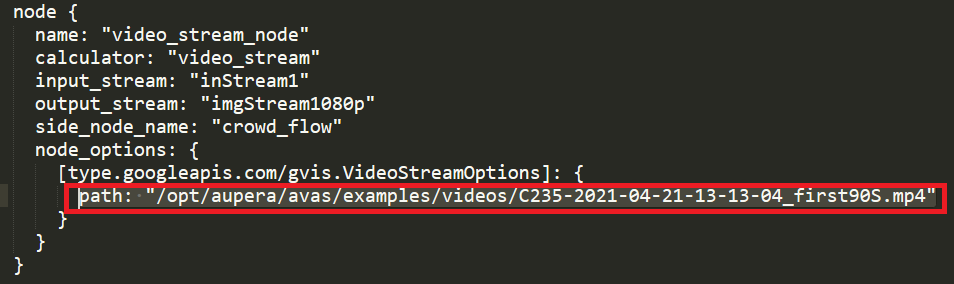
avaser -i input.pbtxt -o output.pbtxt -c using\_rtsp.pbtxt

The results of our examples will be broad case in the IP address specified in the output.pbtxt file. In most cases, this is set to

output\_urls: "rtsp://localhost:8554/out1"

which means that you can see the results by typing above RTSP URL into VLC; replacing “localhost” with the IP address of your VMAccel instance.

**NOTE 1:** If you’d like to run the pipelines on videos other that what we have provided, you will need to modify the “path” parameter in the video\_stream node. As shown in Figure 4.1 below:



**Figure 4.1. Aupera video stream output path in pbtxt file**

**NOTE 2:** If you’d like to try our example pipelines on RTSP streams other than the ones lunched by your VMAccel instance, then you will need to edit the input.pbtxt files to set the input\_rtsp parameter to the URL of your RTSP streams.

For example, if the input.pbtxt of the example you are using contains:

input\_urls: "rtsp://10.10.100.100:8554/stream1"

And the IP address of your VMAccel instance is 99.99.999.999:554/mystream, then you should edit your input.pbtxt to contain the following:

input\_urls: "rtsp://99.99.999.999:554/mystream"

**NOTE 3:** To see the results of example pipeline on RTSP streams in other locations you will need to edit the output.pbtxt file in each folder to point either to the IP address of your RTSP sever; or to a valid file path. For example, if you’re inside the box\_detector example folder, and the IP address of the machine running your RTSP server is 10.10.100.100, then you will need to adjust the output.pbtxt to contain:

output\_urls: "rtsp://10.10.100.100:8554/someKey"

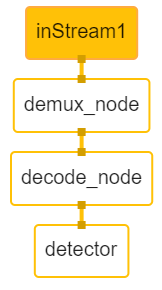
In above case, you can watch the pipeline results at the RTSP stream provided above. Alternatively, your output.pbtxt could include line similar to:

output\_urls: "/tmp/video\_output.mp4"

In above case, the pipeline’s results will be saved to disk in a file accessible via the path specified above.

The pipeline examples that are included with the correct release are as follows:

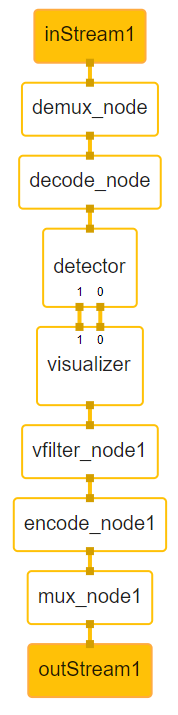
**A. box\_detector/using\_rtsp\_0output.pbtxt**



**Figure 4.2. Aupera pipeline example with demux, decode, and detector nodes**

The pipeline in Figure 4.2 takes one input stream, runs a box\_detector network on the decoded frames, visualizes the detections on the frames, and saves the frames to disk (there is no output video).

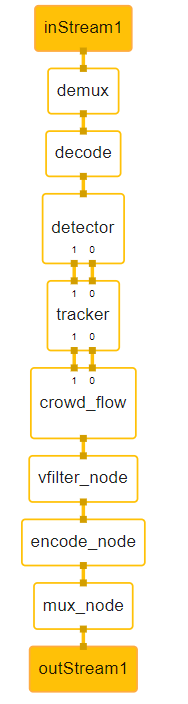
**B. box\_detector/using\_rtsp\_1output.pbtxt**



**Figure 4.3. Aupera pipeline example with multiple nodes 1**

The pipeline in Figure 4.3 takes one input stream and one output stream. It runs a box\_detector network on the decoded frames and sends the detected bounding boxes and the frames to the box\_visualizer node. The box\_visualizer node, will visualize the detected bounding boxes on the frames and send them to video filter, video encoder, and mux nodes. The results are returned in an output rtsp stream or video file.

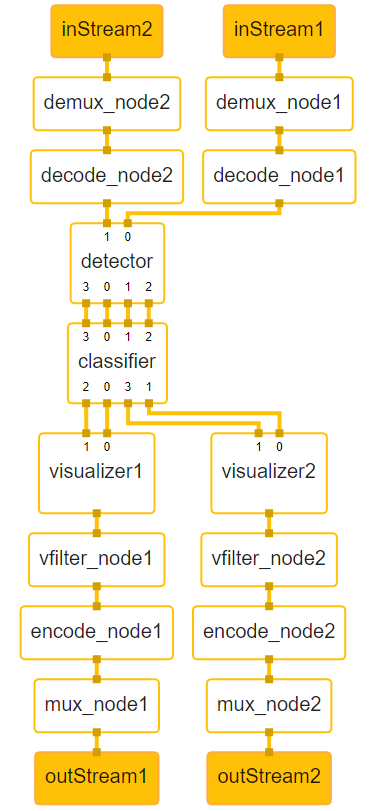
**C. apl\_crowd\_flow/using\_rtsp.pbtxt :**



**Figure 4.4. Aupera pipeline example with multiple nodes 2**

The pipeline in Figure 4.4 takes one input stream and one output stream. It runs a box\_detector (at some interval), which passes the detected bounding boxes and the frames to a box\_Tracker. The box\_tracker tracks the objects (even on frames where the detector has not been run) and sends the bounding boxes and the frames to our crowd\_flow application node. This node applies the crowd\_flow logic; visualizes the results; and passes the frames to the video filter, encode, and mux nodes. The results can be seen in the output rtsp stream or a video file.

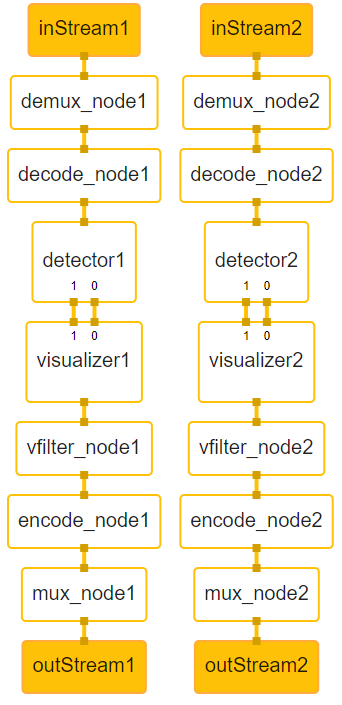
**D. box\_detector\_classifier\_cascade/using\_rtsp.pbtxt**



**Figure 4.5. Aupera pipeline example with two input and output streams**

The pipeline in Figure 4.5 takes two synchronized input streams and produces two output streams. It runs a box detector (at some interval). Then it passes the frames in-tandem with the detections to the classifier node. The classifier node then classifies the objects detected by the detector node. It passes the classifications, which sends the results of each stream to its corresponding box\_visualizer node. The box\_visualizer will overlay the detections and classifications on the frames and send the results to video filter, stream encode, and stream mux nodes to be displayed over RTSP stream.

**E. box\_detector/using\_rtsp.pbtxt**



**Figure 4.6. Aupera pipeline example with two input and output streams**

The pipeline in Figure 4.6 is very similar to example B (figure 4.3); except it runs two detection tasks in parallel. This pipeline takes two input streams and produces two output streams. On stream 1, it runs a box detector with crowd models (head detection); while, on stream 2, it runs a box detector with retail models. The detection results are then passed to corresponding box\_visualizer nodes. The box\_visualizer nodes will overlay the detections on the frames and send the results to video filter, stream encode, and stream mux nodes to be displayed over RTSP stream.

**F. Throughput measurement using retail application**

Similar to VMSS1.0, we use the retail application as an example pipeline for throughput measurement. This application consists of running a TinyYoloV3 object detector along with 3 resnet50 classification networks. All the networks are trained on the objects in the retail scenario. We also use the same retail.mp4 video as before (provided in the example\_videos folder). We maintain (and slightly exceed) the performance of VMSS1.0 by supporting 37 streams (with I-frame-extraction) without any trackers and with 56 streams when using a tracker. We have, however, dramatically improved the accuracy of the pipeline when a tracker is used compared to VMSS1.0.

If you look inside the **example\_pipelines/throughput\_benchmarking/37streams**, you will find 3 configurations:

* **config\_noTracker\_noVideoOut.pbtxt** is the official test configuration. To run this test, you must ensure that the output.pbtxt file is empty.
* **config\_withTracker\_noVideoOut.pbtxt** performs the same test except the tracker is used. Here, we use a cluster size of 5 (i.e. classify each track 5 times) to achieve higher accuracy.
* **config\_withTracker\_withVideoOut.pbtxt** can be used to watch the visualized results on the output stream. To run this config, you will need to use the provided output.pbtxt with the correct output stream paths. Please note that since video visualization is a computationally expensive operation, it is only allowed while running the tracker. Also please note that when I-frame-extraction is true, the output video stream can only be saved at 10fps (since the video has an I-frame every 3rd frame). You can set I-frame\_extraction to false, in order to run and save the video at 30fps but that would require reducing the total number of streams (since we are making the test 3 times harder by passing 3 times the number of frames to the pipeline).

Finally, inside **example\_pipelines/throughput\_benchmarking/56streams** we have provided a single config (**config\_withTracker\_noVideoOut.pbtxt**) which can be used to confirm that the framework can run the retail pipeline with 56 streams without any frame drops. You can increase the **classifications\_cluster\_size** parameter to achieve higher classification accuracy if needed. Although, in this example, even with a value of 1, the classification accuracy is not far lower than running without a tracker. Essentially, this parameter specifies how many times we run the classification on each track. For example, when a value of 5 is specified, for each track, we run the classifier 5 times, and use the most frequent (i.e., the mode) classification as the final result.

# Aupera Node Toolkit

Aupera’s Node Toolkit provides a collection of highly configurable nodes to help users build and launch their own pipelines as quickly as possible. Using VMSS2.0 users no longer have to write any code for tasks that are common to most CV pipelines such as demux, decode, preprocessing, running object detection and classifications, tracking, mux, and encoding. The rest of this section describes the configurations that all VMSS2.0 will support at the pipeline level, configurations that all VMSS2.0 nodes will accept, and a list of nodes currently provided in Aupera Node Toolkit.

## 5.1 Graph-level Configurations

Currently VMSS 2.0 allows for the following configurations at the graph level:

// collection of nodes that comprise the graph  
repeated Node node  
  
//(optional) total number of threads used to execute the graph, if not provided each graph with an execute method will run in its own thread  
int32 num\_threads   
   
// (optional) default max queue size for each node data stream, this can be overridden by individual nodes  
int32 max\_queue\_size  
     
// Names of the input streams for the graph. If using the command line (as opposed to the VMSS 2.0 client), the order of these streams must match the order of the stream URL(s) passed to the server when executing the graph.  
repeated string input\_stream  
    
// Names of the output streams for the graph. If using the command line (as opposed to the VMSS 2.0 client), the order of these streams must match the order of the stream URL(s) passed to the server when executing the graph.  
repeated string output\_stream  
   
// (optional) control port, may be used by the udp/tcp server to communicate runtime control command send by VMSS 2.0 client  
int32 control\_port

## 5.2 Node-level Configurations

Currently VMSS 2.0 allows for the following configurations at the node level:

// (optional) the name of this node used for visualization purposes    
string name   
    
// official name of the node. This needs to match the name the node was registered with and the name of the calculator binary   
string calculator   
    
// names of the input streams to the node. These names must be unique in the graph. Each name (i.e., input stream) also requires either a graph level input stream or a node level output stream with the same name. You can also use the name:tag format to add a tag for display purposes   
repeated string input\_stream   
    
// names of the output streams to the node. These names must be unique in the graph. You can also use name:tag format to use a tag for display purposes   
repeated string output\_stream   
   
// (optional) synchronization mechanism applied to this node by the framework. If not provided, the framework will use a default value of 0  
//The following synchronization modes are currently supported:    
//0: no synchronization the execute method is called as soon as any of the input streams of the node has a packet available    
//1: synchronization using an incremental value (such as pts, gts, or frame number)   
//The frame works will call the execute methods once all the node’s input streams have at least one packet available. The framework will buffer the packets from the earlier streams while waiting for the delayed streams    
//2: sync with incremental value but with reuse of last element   
//The initial behavior of this mode is similar to type 1, except that for subsequent packets the framework will call the execute method of the node as soon as any of the input streams has a packet by cloning the last packet of the input streams    
int32 stream\_sync\_mode   
   
// (optional) maximum number of milliseconds that the framework will on the next packets before forcing the output (i.e., breaking the synchronization promise). By default, this is set to infinity.     
int32 stream\_sync\_maxwait\_ms   
   
// (optional) maximum size of the input queue for this node, this value is set per input stream. Once this value is reached, the framework will no longer enqueue packets for the node until the packets already in the pipeline are consumed (i.e., the input queue size falls below this maximum threshold). This parameter is set to 12 by default.    
int32 queue\_size   
   
// (optional) list of the names of the nodes that this node communicates with via side-packet communication. Both nodes must have each other’s names in their side\_node\_name list to be able to communicate.    
repeated string side\_node\_name   
   
// (optional) the maximum size of the side packet message queue for this node. Once this size is reached, the nodes trying to enqueue messages for this node will receive an error message until this node has consumed at least one of its messages. By default, this parameter is set to infinity.    
int32 max\_side\_queue\_size   
   
// maximum number of threads that this node is allowed to create. Once this limit is reached, the node will be blocked from creating new threads. By default, this parameter is set to infinity.    
Int32 num\_threads   
   
// (optional) this is the list of custom parameters specific to each node. The designer of each node has full freedom in the format (type, names, number) of these parameters as long as they’re supported by protobuf3.   
repeated google.protobuf.Any node\_options

## 5.3 Nodes Currently Included in Aupera Node Toolkit

The following are the nodes that are currently included in the toolkit. Keep in mind that we are constantly adding new nodes. Furthermore, users can very easily create their own VMSS2.0 nodes. For each node, the calculator's name (the unique identifier for this node that must be included in the calculator filed inside of your pipeline pbtxt file) is mentioned as well. To receive documentation on how to create your own nodes you can write to [vmss@auperatech.com](mailto:vmss@auperatech.com).

### 5.3.1 Stream Demux

calculator: “stream\_demux”

The role of this node is to connect to a RTSP stream, receive the packets, and perform demux operation on them. This is the first node in most pipelines and, usually, communicates with a video decoder node. This node currently accepts the following parameters:

// selects between "udp" or "tcp" transport methods  
string   rtsp\_transport  
  
// (optional) normally, the demux node might spend some time (around 1~2s) to calculate the actual fps of an input stream. This option can specify the fps to the demux node to avoid spending this time   
float    force\_fps     
  
// if set to true, the demux node will only send the I frame packet to the decoder, otherwise it will send all packets  
bool   iframe\_extract  
        
// if set to true, then demux node will try to reconnect the input stream when network quality is poor or when eof is received  
bool   auto\_reconnect

### 5.3.2 Video Decoder

calculator: “x86\_decode”

The role of this node is to receive the packets from a demux node and decode them. We currently support X86 (software), AMD/Xilinx Alveo U30 (hardware), and Aupera V205 (hardware) decoding. The decoder can provide the frames in any scale (using hardware scaling is available) and in any color format (using hardware color conversion if available). Currently, h264, h265, and mpeg4 compression formats are supported. This node currently accepts the following parameters:

// decoder name, supports u30\_dec\_h2645 or v205\_dec\_h2645 or x86\_soft\_decoder  
string   name  
  
// support multiple output picture streams with different resolution, if set to 0, then use the same resolution with input stream  
repeated uint32 ow  
repeated uint32 oh  
  
// support multiple output picture streams with different pixel format, supported: "RGB24" or "BGR24" or "NV12" or "I420", default use NV12 as output pixel format  
repeated string opixfmt  
  
//decoder buffer queue size. Once this size is reached, no more frames will be queued.  
uint32   queue\_size  
          
// decoder enable low\_latency or not, it can only be set to true if input stream does not contain B frame. If this is the case, and this option is enabled then decoder will output the frames as soon as possible  
bool    low\_latency

### 5.3.3 Stream Mux

calculator: “stream\_mux”

The role of this node is to receive the encoded frame from a video encoder node and pass it in an RTSP stream. Alternatively, this node can save the output video to a file. This is usually the last node in pipelines that save output videos. The mux node must always be connected to a graph-level output stream. If the server sets this output stream to an IP address, then the result will be transmitted over RTSP. If the server sets this to a file path, then the results will be saved as a video. Note that the server looks at the pbtxt file passed through the -o parameter to the *avaser* command to determine where to set the destination for the mux node. For example, if the command that is running the pipeline is as follows:

avaser -i input.pbtxt -o output.pbtxt -c pipeline.pbtxt

Then, if the output.pbtxt contains valid URLs similar to those below, the results are sent over the RTSP, stream:

output\_urls: "rtsp://10.10.190.128:554/out1"

However, if the output.pbtxt contains a valid file path similar to below, the results is saved to disk:

output\_urls: "/tmp/output\_video.mp4"

This node currently accepts the following parameters:

// selects between "udp" or "tcp" transport methods  
string   rtsp\_transport  
  
// if set to true, then demux node will try to reconnect the input stream when network quality is poor or when eof is received  
bool    auto\_reconnect

### 5.3.4 Video Encode

calculator: “x86\_encode”

The role of this node is to receive the frames and compress them into video packets. As such, this node is usually followed by a mux node. Currently, we support X86 (software), AMD/Xilinx Alveo U30 (hardware), and Aupera V205 (hardware) encoding. Also, h264, h265, and mpeg4 formats are currently supported. This node accepts the following parameters:

// encoder name, support v205\_enc\_h264/v205\_enc\_h265, x86\_enc\_h264/x86\_enc\_h265/x86\_enc\_mpeg4   
string name   
   
// encoder width and height, if set to 0, then use the same resolution with input stream   
uint32 w   
uint32 h   
   
// (optional) the fps of the output video can be supplied. Otherwise, we’ll try to match this to the fps of the input rtsp stream.   
float fps   
   
// encoder buffer queue size. Once this size is reached, no more frames will be queued   
uint32 queue\_size   
   
// if set to true then different B frame number between two P or I frames are assigned  
bool   b\_adapt  
   
// maximum B frame number between two P or I frames   
uint32 bframes  
   
// the interval of two I frames (i.e., for the entire group of frames)   
uint32 gop\_size   
   
// output video bitrate, specified in (bit/s)   
uint32 bitrate   
   
// currently supports "default", "low-latency-B", "low-latency-P", "dynamic"   
string gop\_mode    
   
// currently supports "CBR", "VBR", "CRF"   
string rc\_mode   
   
// can improve the performance of single stream encoding by using multiple threads   
uint32 threads

### 5.3.5 Video Filter

calculator: “ff\_vfilter”

The goal of this node is to adjust the video streaming parameters such as dimensions, fps, color format, etc., as efficiently as possible. This node currently supports the following parameters:

// the pixel format of the output video stream. Such as “I420”, etc.  
string opixfmt;  
  
// width of the output video stream to resize the output frames to. If both this parameter and the oh parameter are provided, roi parameters (roi\_x, roi\_y, roi\_w, and roi\_h) will be ignored.   
uint32 ow;  
  
// height of the output video stream to resize the output frames to. If both this parameter and the ow parameter are provided, roi parameters (roi\_x, roi\_y, roi\_w, and roi\_h) will be ignored.  
uint32 oh;  
  
// fps of the output video stream. If larger than the fps of the input stream, then interpolation is required.  
float  ofps;  
  
// output buffer queue size. Once this size is reached, no more frames will be queued. If not provided infinity is assumed.  
uint32 queue\_size;

### 5.3.6 Object (box) Detector

calculator: “box\_detector”

The role of this node is to run most of the object (box) detectors available on AMD/Xilinx Model Zoo. The user can specify the type (SSD, YoloV2, YoloV3, TinyYolo, RefineNet, faceDetect, etc) and the name of the specific kernel to be used. The assumption is that the model along with its runtime prototxt config (for AMD/Xilinx Model Zoo models, you can find this file provided along with the xmodel, please refer to an example provided [here](https://www.xilinx.com/bin/public/openDownload?filename=yolov3_voc-zcu102_zcu104_kv260-r2.5.0.tar.gz)) file inside is placed on the machine running the object detector node in **/usr/share/vitis\_ai\_library/models** in a folder matching the kernel name inside the AVAS docker.

This node also gives users the ability to determine whether the frames should be returned in the same order that they are received or not. This is controlled by the parameter return\_frames\_in\_order.

If return\_frames\_in\_order = true:

If there is only a single (frame) input stream is provided, then the behavior is controlled based on the provided number of output streams as follows:

* If no output stream is specified, the node will visualize the detected bounding boxes on the frame and write the frame to disk (at the provided task directory path) with the following name: **frame-#\_pts-#\_time-#.jpg.** For example, frame 100 with pts 1000000 with time stamp 9999999 will be saved as frame-100\_pts-1000000\_time-9999999.jpg
* If there is at least one output stream is provided, the detected bounding boxes will be returned as a stream of VinfMetaData objects.
* If there are two output streams provided, the first input stream will contain the detections (in the form of VinfMetaData), and the second input stream will contain the frames. The frames will not be modified in any way (i.e., no drawing of the detected bounding boxes).

If there several (frame) input streams provided, then the behavior is controlled based on the provided number of output streams as follows:

* If no output stream is specified, the behavior is the same as above.
* If only a single output stream is specified, a runtime error is thrown
* If the specified number of outputs is the same as the specified number of inputs, then the detected bonding boxes of each stream are returned in a VinfMetaData stream. The order of the output streams will be the same as the input streams.
* If the specified number of outputs is exactly double the specified number of inputs, then aside from the detected bounding boxes that are returned in VinfMetaData streams, the original frames are returned as well. For example, if there are 5 inputs streams and 10 output streams, then the outputs streams 1 to 5 will contain the detected bounding boxes (i.e., VinfMetaData streams) of the input streams 1 to 5 respectively, and output streams 6 to 10 will contain the original frames of input streams 1 to 5 respectively. In other words, the detected bounding boxes of input stream index n (with index 0 corresponding to the first input stream) will be returned in output stream index n, while the frames of input stream index n will be returned in output stream index (n + number of input streams).

If return\_frames\_in\_order = false:

If there is only a single (frame) input stream is provided, then the behavior is the same as above (based on number of outputs) aside from the fact that there are no guarantees on the order with which the frames and detections are returned.

If more than one input stream is provided, then the behavior is controlled based on the provided number of output streams as follows:

* If no output stream is specified, the node will visualize the detected bounding boxes on all the frames of all the streams and write the frame to disk (at the provided task directory path) with the following name: **stream-#\_frame-#\_pts-#\_time-#.jpg.** For example, frame 100 from the fifth stream with pts 1000000 with time stamp 9999999 will be saved as **stream-5\_frame-100\_pts-1000000\_time-9999999.jpg**
* If there is at least one output stream is provided, the detected bounding boxes of all the streams will be returned as a stream of VinfMetaData objects. In this way, the frames from different input streams will be merged into a single VinfmetaData output stream (i.e., neither the frames nor the streams will be in order).
* If two output streams are provided, aside from the detections (in the form of VinfMetaData), the input streams (frames) are returned as well. The frames will not be modified in any way (i.e., no drawing of the detected bounding boxes). In this case, the frames of all the input streams will be merged into a single output frame stream.
* If the number of output streams is equal to the number of input streams + 1, then aside from the VinfMetaData streams containing the detected bounding boxes, all the input streams are returned in their original format (i.e., in individual streams as well).

This node accepts the following parameters:

// the message format specifying the pixel mean subtracted from all the pixels of the input frame        
message Mean {    
        float r;    
        float g;    
        float b;    
  }    
     
// the message format specifying the scale multiplied to all pixels of the input frame        
message Scale {    
        float ch1;    
        float ch2;    
        float ch3;    
}    
    
// the message format specifying the detection confidence threshold applied to each of the classes for the network    
 message LabelConfidence {    
        int32 label;    
        float confidence;    
}    
    
// the message format specifying the nms (none max suppression) threshold applied to each of the classes for the network    
message InterClassNms {    
        float threshold;    
        repeated int32 labels;     
}    
    
// pixel mean subtracted from all the pixels of the input frame    
Mean mean;    
    
// pixel scale multiplied to all pixels of the input frame        
Scale scale;    
    
// detection confidence threshold applied to each of the classes for the network    
repeated LabelConfidence label\_confidence;    
    
// the nms (none max suppression) threshold applied to each of the classes for the network    
repeated InterClassNms inter\_class\_nms;    
    
// detection interval, value of 1 means the network is run on every frame, value of 2 means the detection is run on every other frame, value of 3 means the detection is run on every third frame, etc    
int32 detect\_interval;    
    
// the type of the detector to be used, we currently support SSD, YoloV2, YoloV3, TinyYolo, RefineNet, FaceDetect, and several others    
string detector\_type;    
    
// name of the specific kernel to be used. The assumption is that the model is placed on the machine running the object detector node in /usr/share/vitis\_ai\_library/models in a folder matching the kernel name    
string kernel\_name;    
    
// in case of an obfuscated network, this will be the obfuscated string token (in this case, kernel\_name parameter is not needed)    
string obfuscated\_token;    
    
// whether the node should resize the input frame to the input dimensions of the network. This is set to true in most cases, unless the frame dimensions happen to be exactly what the network requires    
bool need\_preprocess;    
    
// logs timing (latency) information    
bool log\_performance;    
    
// whether to letter box the input image or not. If set true, letterboxing is done while maintaining the aspect ratio of the original frame    
bool run\_on\_letterboxed\_img;    
    
// when set to true, the black frames are ignored (the frame\_number is not incremented, and these frames are not passed to the detector). This is when the rtsp stream is from a video that has black padding at its beginning or end. Usually, this is used when trying to match the performance on a video to that of the rtsp stream of the same video.    
bool ignore\_black\_frames;    
    
// batch size to use for running the model, if not supplied batch size of 1 is assumed. If the provided value is larger than the batch size that the hardware supports, this value is capped (to what the hardware supports)    
int32 batch\_size;    
   
// if set to true a batch is only passed to the detector once its size reaches the specified batch size (or when the batch\_collection\_timeout\_ms is reached). If set to false, frames are passed to the detector as soon as they arrive (so the specified batch size is only applied if anough frames have been collected while the detector is processing the previous batch).  
bool force\_batch\_size;  
   
// number of milli-seconds the node will wait on a batch of frames to be collected. Once this number is reached, the collected batch is sent to the detector regardless of its size.   
int32 batch\_collection\_timeout\_ms;   
   
// if true, then the ordering of the frames within the same stream and the ordering of the streams will be maintain (i.e., the output streams will follow the same ordering). When using a single thread, the ordering is always maintained. However, when using multiple threads, not maintaining the order (setting this parameter to false) will improve performance.   
bool return\_frames\_in\_order;   
    
//number of threads to be used for running a detection. Multi-threaded processing loses the order of the frames (and the order of the streams); so if in order processing is required, the process\_frames\_in\_order must be set to true. Single threaded detection will always maintain the original ordering (of the frames and streams). If this parameter is not provided value of 1 is assumed.    
int32 detection\_threads;

### 5.3.7 Object Tracker

calculator: “box\_tracker”

The goal of this node is to support multi-object tracking on a single stream. We support several types of trackers including Sort++, DeepSort. DeepSort, requires an encoding of the objects to be provided; as such, it’s the most accurate tracking method available in this node. However, accuracy and latency are controlled by the encoder network (i.e., it requires the encodings to perform similarity matching, which needs to be run outside of the tracker). Sort++, on the other hand, does not require encoding of the objects (to perform similarity) which makes it the fastest tracker supported by this node (with descent accuracy in most conditions).

This node has the capability of receiving the detect\_interval that the detector node uses via side packets. In this case, on frames that the detector has not been run, the tracker will make predictions as to where the objects are. On frames, that detector is run, the tracker will use the detected bounding boxes to correct its estimation models.

The first input stream of this node is always assumed to be of the type VinfMetaData containing the detected bounding boxes. When using DeepSort, the first stream must contain the encodings of the detected objects as well. If present, the second input stream is assumed to be the video stream (of frames) corresponding to the detected bounding boxes. If two streams are provided, then the synchronization flags are mandatory to ensure that the detected bounding boxes (from the first stream) correspond to the frame (from the second stream) if within the detect interval.

This node only provides single stream/camera tracking, for multi streams tracking you should use the provided multi stream tracker (i.e., reid) node instead. This node will be provided in our future release.

The logic for the output of the tracker node is as follows:

* If no output stream is specified, but a second input stream (i.e., video frames) is provided, the node will visualize the detected (and predicted) bounding boxes on the frame and write the frame to disk (at the provided task directory path) with the following name: **frame-#\_pts-#\_time-#.jpg.** For example, frame 100 with pts 1000000 with time stamp 9999999 will be saved as **frame-100\_pts-1000000\_time-9999999.jpg**
* If no output stream is specified, and there is only a single input stream (i.e., just the detected bounding boxes) is provided, a runtime error is thrown.
* If more than one output stream is specified, the first one will always be of the type VInfMetaData containing the detected (and predicted) bounding boxes. If there is more than one input streams (i.e., a stream of frames is also supplied as input) then the second output stream will contain the input frames without any modifications.

This node accepts the following parameters:

// type of the tracker to be used currently we support Sort++, and DeepSort among others  
string tracker\_type;  
  
// predicted bounding boxes with areas smaller than this threshold (in terms of square pixels) will be ignored  
int32 min\_object\_area\_th;  
  
// speed buffer parameter of the MOSSE tracker (not used by Sort++ or DeepSort trackers)  
int32 speed\_buffer\_max\_size;  
   
// Maximum number of detect intervals that the tracker will keep a track that does not get matched with any new detections alive  
int32 max\_keep\_alive;  
   
//minimum number of matched detections required before a track is considered reliable  
int32 min\_hits  
  
// used by sort++ and deepSort trackers. If a detected object achieves a total affinity score that is higher than this threshold with a tracker, it will be considered a match (will be used update the tracker state).  
float affinity\_threshold;  
  
// used by sort++ and deepSort trackers. Weight of the shape similarity score inside the total affinity score  
float shape\_weight;  
  
// used by sort++ and deepSort trackers. Weight of the position similarity score inside the total affinity score  
float position\_weight;  
  
// used by sort++ and deepSort trackers. Weight of the appearance (i.e., encoding vector) similarity score inside the total affinity score  
float appearnace\_weight;  
  
// used by sort++ and deepSort trackers. If the shape similarity score between a detected bounding box and a tracker is larger than this threshold, a match will not happen (regardless of all other scores)       
float shape\_dist\_max;  
  
// used by sort++ and deepSort trackers. If the position similarity score between a detected bounding box and a tracker is larger than this threshold, a match will not happen (regardless of all other scores)   
float position\_dist\_max;  
  
// used by sort++ and deepSort trackers. Toggling between exp cost and weighted sum cost      
bool use\_exp\_cost;

### 5.3.8 Image Classifier

calculator: “box\_classifier”

The role of this node is to run any of the classifier networks available on Xilinx model zoo. The user can specify the type (ResNet, Inception, SqueezeNet, etc.) and the name of the specific kernel to be used. The assumption is that the model is placed on the machine running the object detector node in **/usr/share/vitis\_ai\_library/models** in a folder matching the kernel name.

If the only a single (frame) input stream is provided, containing full frames of cropped detected objects (that the classifier must run on), then the behavior is controlled based on the provided number of output streams as follows:

* If no output stream is specified, the node will save the frames (or cropped images) with the detected class in the name according to the following format: **stream-#\_frame-#\_pts-#\_time-#.jpg.** For example, the fifth crop of frame 100 with pts 1000000 with time stamp 9999999, with the detected class “car” will be saved as **frame-100\_pts-1000000\_time-9999999.jpg**. Bounding boxes and the labes are overlayed on the image.
* If there is at least one output stream is provided, the detected classes will be returned as a stream of VinfMetaData objects.
* If more than one output stream is provided, aside from the classes (in the form of VinfMetaData), the input stream (crops or frames) is returned as well. The frames (or crops) will not be modified in any way (i.e., no drawing of the detected bounding boxes).

If more than one input stream is provided, then the user must specify whether they require the resulting classed to be returned in separate streams or not using the **return\_in\_order** parameter. If this parameter is set to true, then:

* If no output stream is specified, the node will save the frames (or cropped images) with the stream index and detected class in the name according to the following format: **stream-#\_frame-#\_pts-#\_time-#.jpg.** For example, the fifth crop of frame 100 from 6th stream with pts 1000000 with time stamp 9999999, with the detected class “car” will be saved as **crop-5\_class\_car\_stream-6\_frame-100\_pts-1000000\_time-9999999.jpg**
* If the number of outputs is equal to number of image input streams, then classifier node sends classifications in the form of VinfMetaData to the output streams. There is a one-to-one relationship between the input frames streams and output streams
* If the number of output streams is equal to twice the number of image input streams, then aside from the VinfMetaData streams containing the detected classes boxes, all the input streams are returned in their original format (i.e., in individual streams as well).

If the return\_in\_order parameter is set to true, then:

* If no output stream is specified, the behavior is the same as above.
* If only a single output stream is specified, a runtime error is thrown
* If the specified number of outputs is the same as the specified number of inputs, then the detected classes of each stream are returned in a VinfMetaData stream. The order of the output streams will be the same as the input streams.
* If the specified number of outputs is exactly double the specified number of inputs, then aside from the detected classes that are returned in VinfMetaData streams, the original frames (or crops) are returned as well. For example, if there are 5 inputs streams and 10 output streams, then the first 5 outputs streams are the detected bounding boxes (i.e., VinfMetaData streams) and the last 5 outputs streams are the original frames.

This node accepts the following parameters:

// the type of the classifier to be used, we currently support ResNet, Inception, SqueezeNet, and several others  
string classifier\_type;  
  
// name of the specific kernel to be used. The assumption is that the model is placed on the machine running the object classifier node in /usr/share/vitis\_ai\_library/models in a folder matching the kernel name  
string kernel\_name;  
  
// whether the node should resize the input frame to the input dimensions of the network. This is set to true in most cases, unless the frame dimensions happen to be exactly what the network requires  
bool need\_preprocess;  
  
// whether to letter box the input image or not. If set true, letterboxing is done while maintaining the aspect ratio of the original frame  
bool run\_on\_letterboxed\_img;  
  
// batch size to use for running the model, if not supplied batch size of 1 is assumed. If the provided value is larger than the batch size that the hardware supports, this value is capped (to what the hardware supports)  
int32 batch\_size;  
  
// if true, then the ordering of the frames (or crops) within the same stream and the ordering of the streams will be maintain (i.e., the output streams will follow the same ordering). When using a single thread, the ordering is always maintained. However, when using multiple threads, not maintaining the order (setting this parameter to true) will improve performance.  
bool return\_in\_order;  
  
// number of threads to be used for running a classification. Multi-threaded processing loses the order of the frames (and the order of the streams); so if in order processing is required, the process\_frames\_in\_order must be set to true. Single threaded classification will always maintain the original ordering (of the frames and streams). If this parameter is not provided value of 1 is assumed.    
int32 classification\_threads;  
  
// whether batch sizes are forced  
bool force\_batch\_size;  
  
// the timeout of each batch. If batching takes up to this time even though batch\_size is not reached, the batch will be submitted  
uint64 batch\_collection\_timeout\_ms;  
  
// whether or not classifier uses detections as inputs  
bool use\_detections;  
  
// logs timing (latency) information  
bool log\_performance;  
  
// maximum size of queue that classifier library uses  
int32 max\_classification\_lib\_q\_size;  
  
// maximum allowed cache size for frames. This option is meaningful when force\_batch and return\_in\_order is true. In the case that there are no detections in several consecutive frames of the input, this prevents the output to be too far behind the input. The default value is 8  
  
int32 max\_frame\_cache\_size;

### 5.3.9 Object (box) Visualizer

calculator: “box\_visualizer”

The goal of this need is to visualize the output of Object (box) Detector, Object tracker, and Image Classifier nodes. This node has different versions that use hardware accelerated methods (on platforms that support this functionality) and software (OpenCV) methods on platforms that don’t support hardware acceleration. Based on these options, the behavior is as follows:

* If input\_type is INPUT\_TYPE\_DETECTION, then the assumption is that the first input stream is of the type VinfMetaData containing the detected bounding boxes while the second input stream contains the frames.

In this case, if there is no output stream specified, then the frames with the bounding boxes visualized on them will be written to disk according to **frame-#\_pts-#\_time-#.jpg.** For example, frame 100 with pts 1000000 with time stamp 9999999 will be saved as frame-100\_pts-1000000\_time-9999999.jpg.

If an output stream is specified, then it will contain the frames with the detected bounding boxes visualized on them.

* If input\_type is INPUT\_TYPE\_CLASSFICATION, then the assumption is that the first input stream is of the type VinfMetaData containing the detected classifications while the second input stream contains the frames (or the crops) that generated the classifications.

In this case, if there is no output stream specified, then the frames (or the crops) will be written to disk according to **crop-#\_class-str\_frame-#\_pts-#\_time-#.jpg.** For example, the fifth crop of frame 100 with pts 1000000 with time stamp 9999999, with the detected class “car” will be saved as **crop-5\_class\_car\_frame-100\_pts-1000000\_time-9999999.jpg**

If an output stream is specified, then it will contain the frames (or crops) with the classifications and their bounding boxes (in the case that it applies) overlaid on them starting from the top left corner of the frame (or crop) + an offset.

This node accepts the following parameters:

// the message format specifying the color to be used for drawing bounding boxes or text  
message Color {  
 float r;  
 float g;  
 float b;  
}  
   
// the message format specifying color to be used for specific classes  
message ClassColor {  
 Color color;  
 int32 label;  
}  
   
// the message format specifying the offset to be used for the location of drawing texts  
message Offset {  
 float x;  
 float y;  
}  
   
// this is a type that determines the input type of the node. indicating if it is detections or classifications  
enum InputType {  
 INPUT\_TYPE\_DETECTION;  
 INPUT\_TYPE\_CLASSIFICATION;  
}  
   
// this is the variable that determines the input type based on the enum type above  
InputType input\_type;  
   
// This determines the color of the box in the case that input\_type == INPUT\_TYPE\_DETECTION  
Color box\_color;  
   
// This declares an array of colors for classifications. Each element of this repeated message indicates what color that specific classification should be have. This color is both used for the text and the bounding box. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
repeated ClassColor class\_color;  
   
// In the case that some classification color for a specific class is not defined in class\_color array, the color will default to this. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
Color default\_class\_color;  
   
// Determines the offset of the text from the top-left corner of the bounding box. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
Offset text\_offset;  
   
// Determines the thickness of the box. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
int32 box\_thickness;  
   
// Determines the text\_size. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
int32 text\_size;  
   
// This determines the font of the text for classifications according to HersheyFonts. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
int32 font;  
   
// This determines the font size. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
double font\_scale;  
   
// This determines the font thickness. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
int32 font\_thickness;  
   
// This determines the line\_type used in classification text. the line\_type is according to #LineTypes. This option is only valid if input\_type == INPUT\_TYPE\_CLASSIFICATION.  
int32 line\_type;

### 5.3.10 Image Stream calculator

calculator: “image\_stream”

The goal of this node is to give the ability to the user to stream a set of images instead of a live camera/RTSP stream. This gives the flexibility for testing certain scenarios as well as pipelines that require a directory of images.

This node has a single input which is a dummy stream address. This address is not used anywhere so best is to use some arbitrary dummy rtsp stream that does not exist.

This node has a single output that similarly to decode node outputs a stream of GvisVframes. The images are iterater through in alphabetical order. After the last image is reached, then stream restarts from the first file.

Input\_type determines the type of frames to be used as input.

// this integer determines the frame interval between each image. for example for 25 fps, the value should be set to 40.  
uint32 frame\_interval\_ms;  
   
// this is the directory in which the images are stored  
string directory;  
   
// This is the target width of the image. If the image width is not as specified here, it will be scaled to fit it  
uint32 width;  
   
// This is the targer height of the image. If the image height is not as specified here, it will be scaled to fit it  
uint32 height;  
   
// This determines the type of images in the input  
enum InputType {  
  // Raw Vooya images  
  VOOYA\_BGR = 0;  
  // jpeg images  
  JPEG = 1;  
  // png images  
  PNG = 2;  
}  
   
// instantiates InputType to fit the images in the directory  
InputType input\_type;

### 5.3.11 Video Stream calculator

calculator: "video\_stream”  
The goal of this node is to use a video file to create a stream of images instead of RTSP streams. This gives flexibility for testing pipelines when there is no camera or RTSP stream or simply when the pipeline wants to use a video file as input

This node has a single input which is a dummy stream address. This address is not used anywhere so best is to use some arbitrary dummy rtsp stream that does not exist.

This node has a single output that similarly to decode node outputs a stream of GvisVframes. After the last frame of the video is reached, then stream restarts from the beginning of the video file.

Like the demux node, this node has a side packet which sends out information like fps, dimensions, and more.

// If set to 0, it will use the videos fps. If not, slow-down/speed-up the video to match the speed  
float playback\_fps;  
   
// this indicates the path to the video file  
string path;  
   
// This is the target width of the image. If the image width is not as specified here, it will be scaled to fit it. If either of width/height is not defined or is equal to 0, then original video width will be used  
uint32 width;  
   
// This is the targer height of the image. If the image height is not as specified here, it will be scaled to fit it. If either of width/height is not defined or is equal to 0, then original video height will be used  
uint32 height;