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 5

1.1 VMSS2.0 PIPELINES 8

2 RUNNING VMSS2.0 ON VMACCEL 13

2.1 LAUNCHING AUPERA VMSS2.0 INSTANCE ON VCK5000 13

2.2 USING VMSS2.0 WEB CLIENT 14

2.2.1 RUNNING AUPERA'S CROWD FLOW APPLICATION 14

2.2.2 RUNNING CUSTOM PIPELINES 22

2.3 USING VMSS2.0 SERVER (VIA COMMAND LINE) 29

2.3.1 VMSS2.0 SERVER DOCKER 29

2.4 LAUNCHING YOUR OWN RTSP STREAMS 32

3 RUNNING VMSS2.0 ON-PREMISES 34

3.1 PREREQUISITES 34

3.2 VMSS2.0 SERVER (AVAS) 34

3.3 SETTING UP RTSP STREAMS 35

4 VMSS2.0 PIPELINES 38

4.1 RUNNING VMSS2.0 PIPELINES 38

4.2 PIPELINE EXAMPLES 38

5 AUPERA NODE TOOLKIT 47

5.1 GRAPH-LEVEL CONFIGURATIONS 47

5.2 NODE-LEVEL CONFIGURATIONS 48

5.3 NODES CURRENTLY INCLUDED IN AUPERA NODE TOOLKIT 50

5.3.1 STREAM DEMUX 50

5.3.2 VIDEO DECODER 51

5.3.3 STREAM MUX 51

5.3.4 VIDEO ENCODE 52
5.3.5 VIDEO FILTER 53
5.3.6 OBJECT (BOX) DETECTOR 54
5.3.7 OBJECT TRACKER 59
5.3.8 IMAGE CLASSIFIER 61
5.3.9 OBJECT (BOX) VISUALIZER 64
5.3.10 IMAGE STREAM CALCULATOR 67
5.3.11 VIDEO STREAM CALCULATOR 68
6 CREATING VMSS2.0 PIPELINES 71
6.1 MAJOR OPERATIONS SUPPORTED AT GRAPH LEVEL 71
6.2 SYNCHRONIZATION AND ORDERING 72
6.3 SIDE-PACKET COMMUNICATIONS 73
6.4 INPUT/OUTPUT STREAMS 74
6.5 DATA STREAMS 75
6.6 GRAPH LEVEL CONFIGURATIONS 76
6.7 NODE LEVEL CONFIGURATIONS 77
7 CREATING VMSS2.0 NODES 83
7.1 STEPS FOR CREATING VMSS2.0 NODES 83
TO CREATE YOUR OWN VMSS2.0 NODE YOU NEED TO PERFORM THE 6 STEPS BELOW: 83
7.4 NODE INITIALIZATION 87
7.5 DEFINING THE EXECUTE METHOD 88

1 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, which enables stream-aware processing.

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.

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

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.

1.1 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, including arbitrary graph types, back edges, and side packets, added in VMSS 2.0.

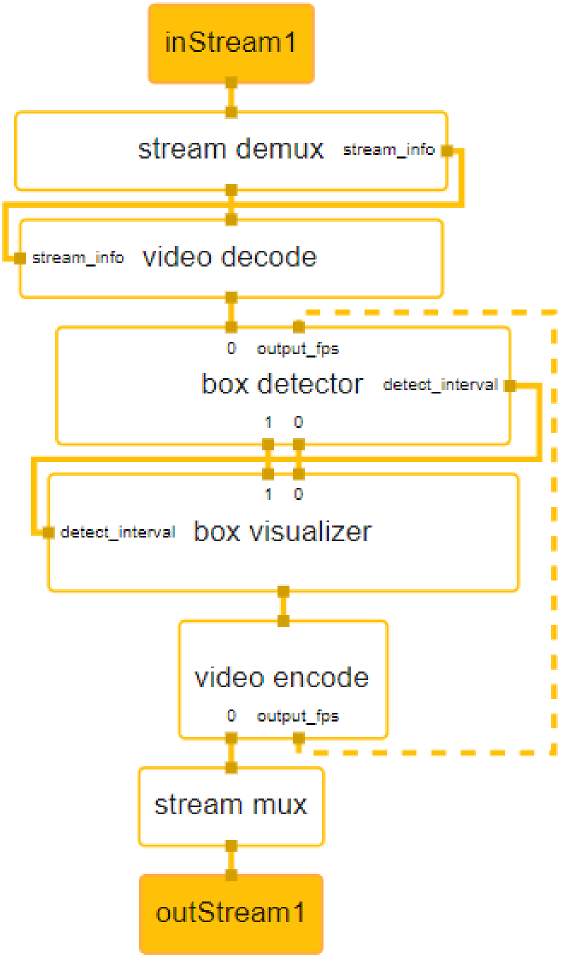

inStream1
stream demux
stream_info
stream_info
video decode
0
output_fps
box detector
detect_interval
1
0
1
0
detect_interval
box visualizer
video encode
0
output_fps
stream mux
outStream1

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), 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"
}
```

2 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.

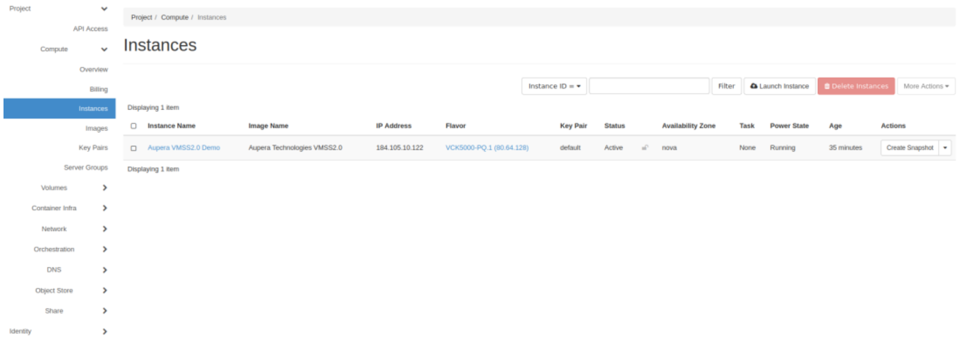
2.1 Launching Aupera VMSS2.0 instance on VCK5000

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.


Project
API Access
Compute
Overview
Billing
Instances
Images
Key Pairs
Server Groups
Volumes
Container Infra
Network
Orchestration
DNS
Object Store
Share
Identity
Project / Compute / Instances
Instance ID = ▾
Filter
Launch Instance
Delete Instances
More Actions ▾

Instances

Displaying 1 item

☐	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
☐	Aupera VMSS2.0 Demo	Aupera Technologies VMSS2.0	184.105.10.122	VCK5000-PQ.1 (80.64.128)	default	Active	nova	None	Running	35 minutes	Create Snapshot ▾

Displaying 1 item

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.

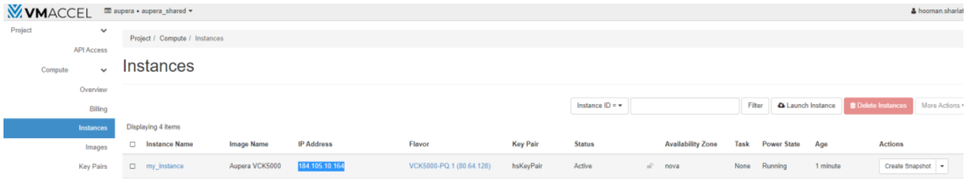
NOTE 1: To run VMSS2.0 pipelines, you can either use Aupera’s VMSS2.0 Web Client (as described in section 2.2) or access the VMSS.2.0 server (as described in section 2.3) using the command line.

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.


VMACCEL
aupera • aupera_shared
hooman.shariat
Project
API Access
Compute
Overview
Billing
Instances
Images
Key Pairs
Project / Compute / Instances
Instances
Instance ID =
Filter
Launch Instance
Delete Instances
More Actions

Displaying 4 items

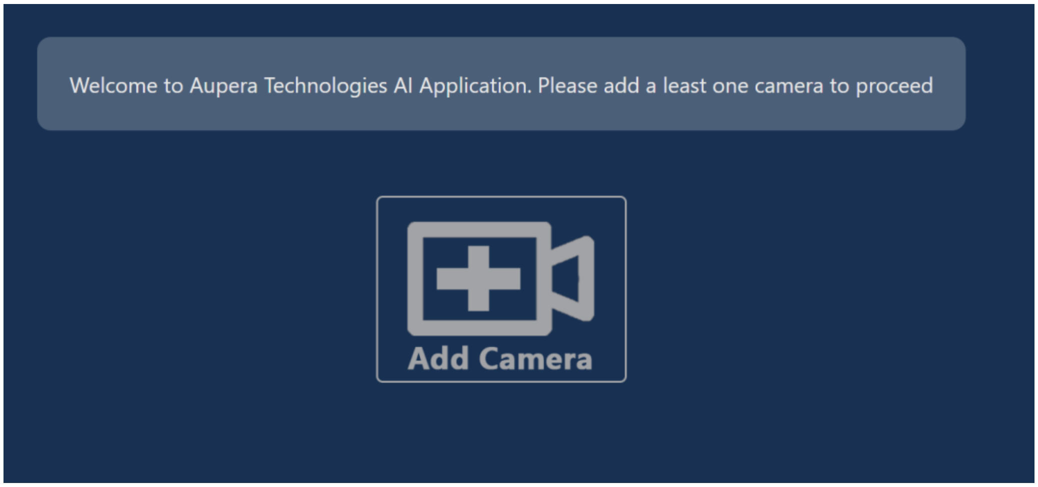
☐	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
☐	my_instance	Aupera VCK5000	184.105.10.164	VCK5000-PQ.1 (80.64.128)	hsKeyPair	Active	nova	None	Running	1 minute	Create Snapshot

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

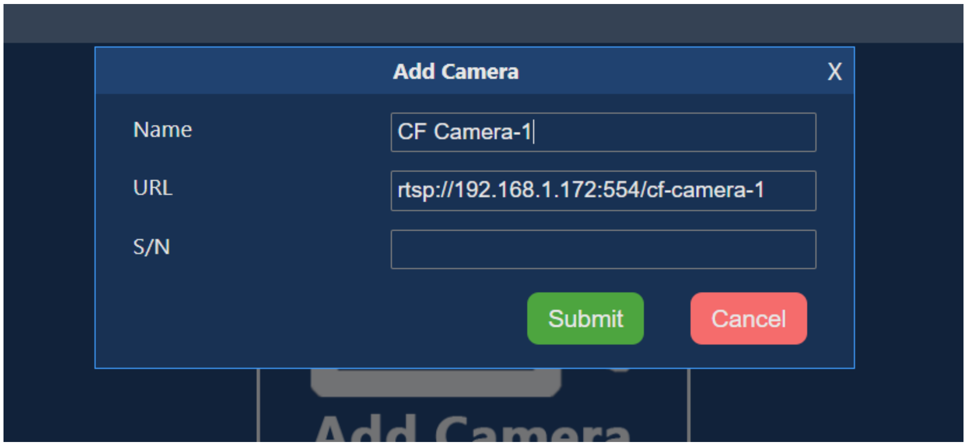
A. Click ***Add Camera*** to add at least one camera to access the Application functions.



Welcome to Aupera Technologies AI Application. Please add a least one camera to proceed

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.



Name: CF Camera-1

URL: rtsp://192.168.1.172:554/cf-camera-1

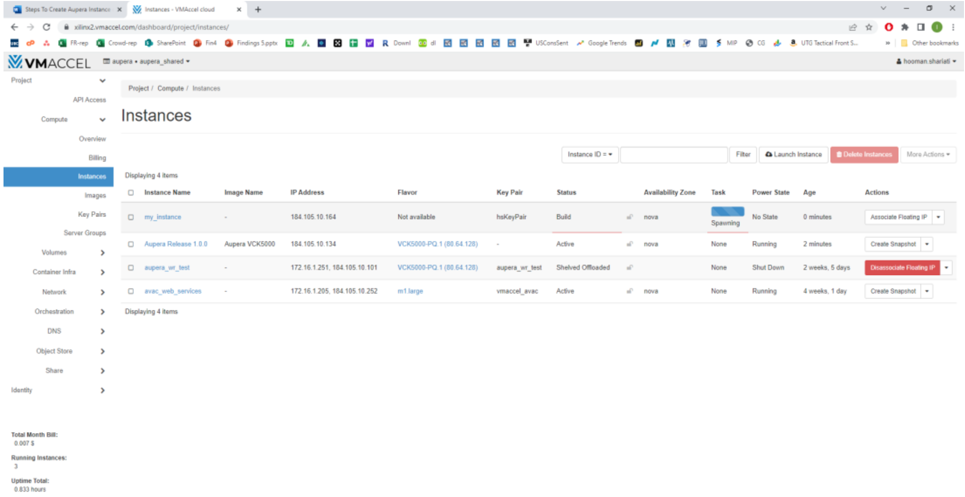
S/N:

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 (**URL**)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.


VMACCEL
aupera • aupera_shared
hooman shariati
Project
API Access
Compute
Overview
Billing
Instances
Images
Key Pairs
Server Groups
Volumes
Container Infra
Network
Orchestration
DNS
Object Store
Share
Identity
Project / Compute / Instances
Instance ID =
Filter
Launch Instance
Delete Instances
More Actions

Instances

Displaying 4 items

☐	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
☐	my_instance	-	184.105.10.164	Not available	hsKeyPair	Build	nova	Spawning	No State	0 minutes	Associate Floating IP
☐	Aupera Release 1.0.0	Aupera VCK5000	184.105.10.134	VCK5000-PQ.1 (80.64.128)	-	Active	nova	None	Running	2 minutes	Create Snapshot
☐	aupera_wr_test	-	172.16.1.251, 184.105.10.101	VCK5000-PQ.1 (80.64.128)	aupera_wr_test	Shelved Offloaded		None	Shut Down	2 weeks, 5 days	Disassociate Floating IP
☐	avac_web_services	-	172.16.1.205, 184.105.10.252	m1.large	vmaccel_avac	Active	nova	None	Running	4 weeks, 1 day	Create Snapshot

Displaying 4 items

Total Month Bill:
0.007 $
Running Instances:
3
Uptime Total:
0.833 hours

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.

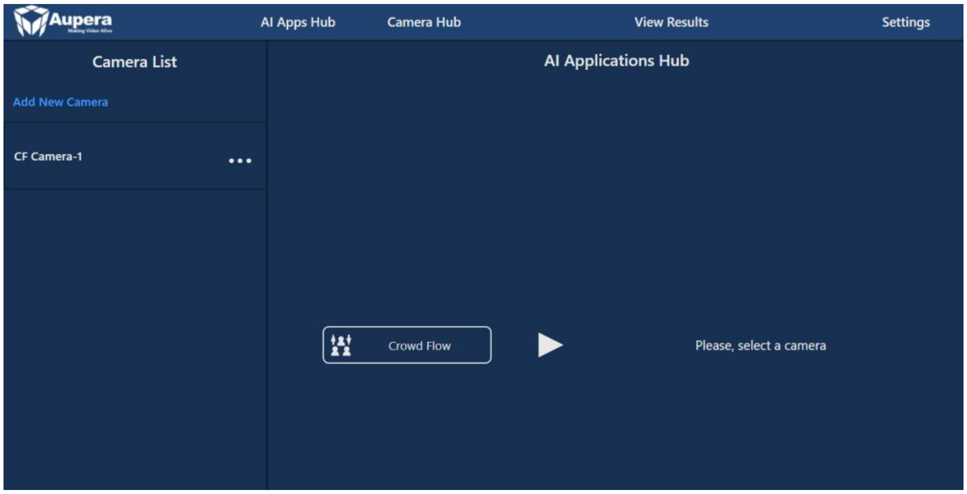

Aupera
AI Apps Hub
Camera Hub
View Results
Settings
Camera List
Add New Camera
CF Camera-1
AI Applications Hub
Crowd Flow
Please, select a camera

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

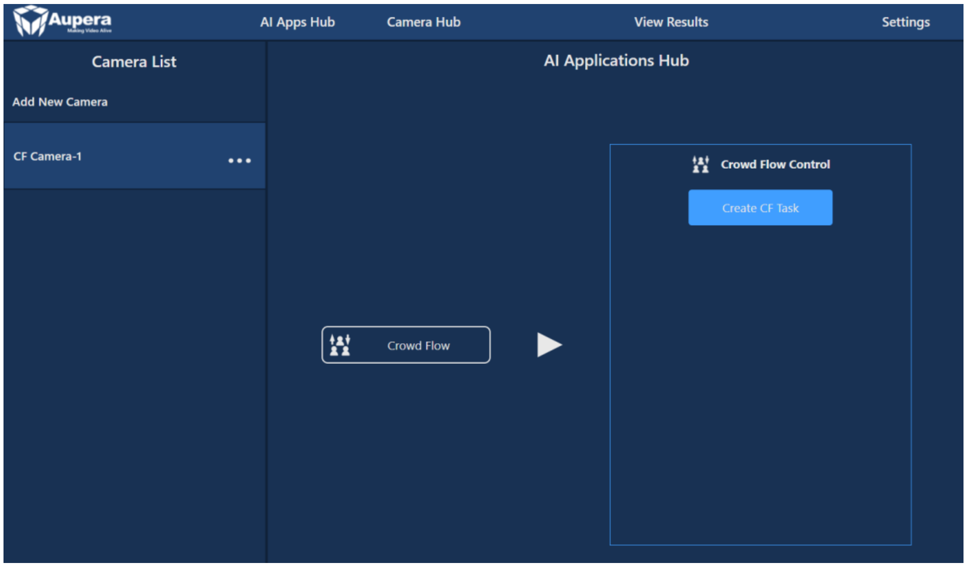
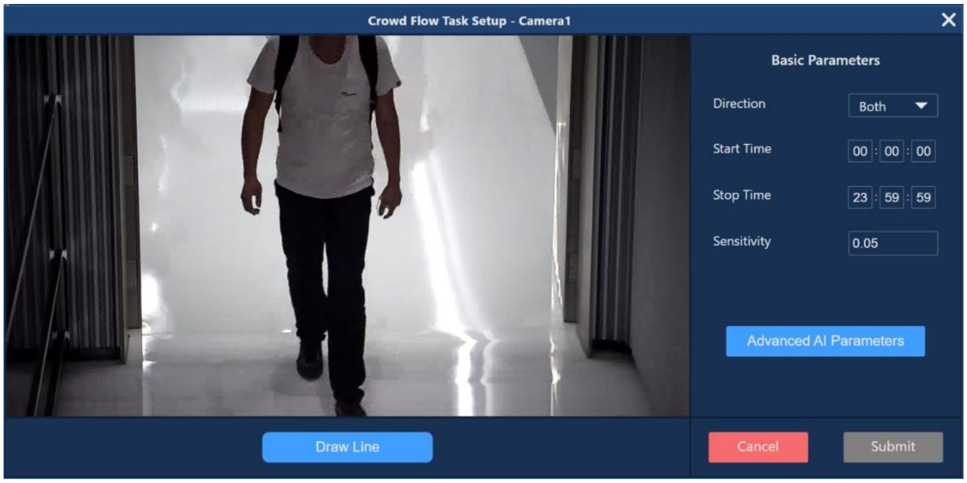

Aupera
AI Apps Hub
Camera Hub
View Results
Settings
Camera List
Add New Camera
CF Camera-1
AI Applications Hub
Crowd Flow Control
Create CF Task
Crowd Flow

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.


Crowd Flow Task Setup - Camera1
Basic Parameters
Advanced AI Parameters
Draw Line
Cancel
Submit

Direction: Both
Start Time: 00 : 00 : 00
Stop Time: 23 : 59 : 59
Sensitivity: 0.05

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.

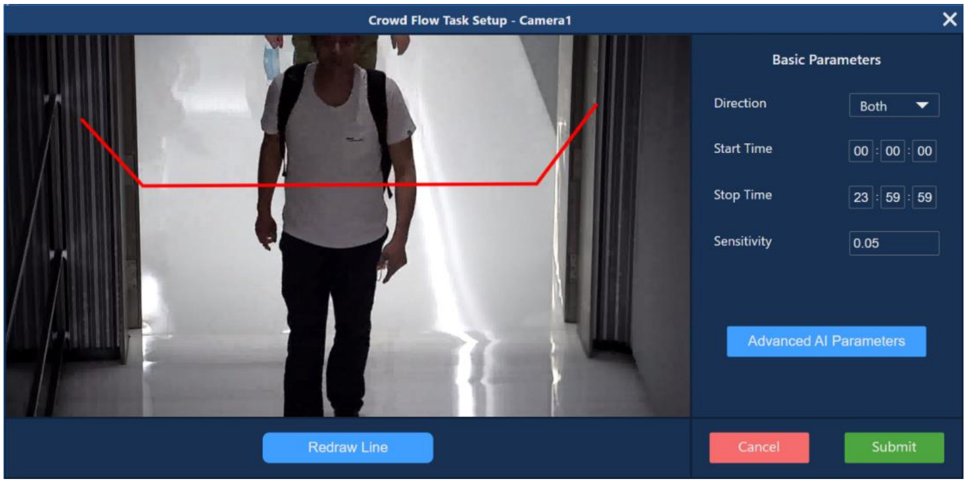
2. **Left-click** and **hold the mouse button** on the place where you would like to start the line.

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.



Direction: Both

Start Time: 00 : 00 : 00

Stop Time: 23 : 59 : 59

Sensitivity: 0.05

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

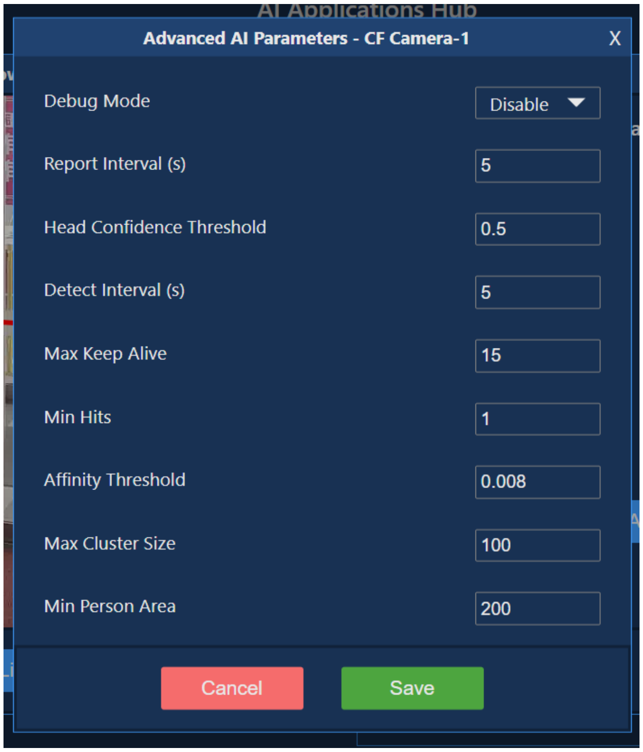
F. Changing Basic Parameters is not required to start the task, you can keep default values.

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.

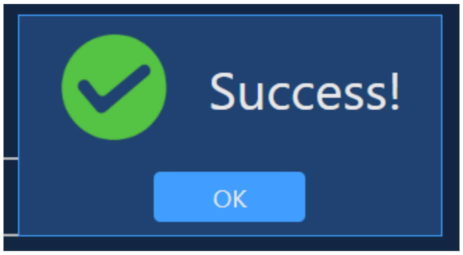


Advanced AI Parameters - CF Camera-1

Debug Mode: Disable
Report Interval (s): 5
Head Confidence Threshold: 0.5
Detect Interval (s): 5
Max Keep Alive: 15
Min Hits: 1
Affinity Threshold: 0.008
Max Cluster Size: 100
Min Person Area: 200

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

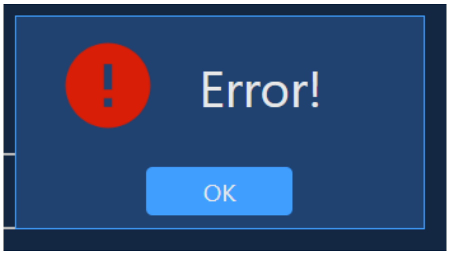
H. To start the task, click the ***Submit*** button. After that, a pop-up message will notify you that the task was successfully launched.



Success!

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.



Error!

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.

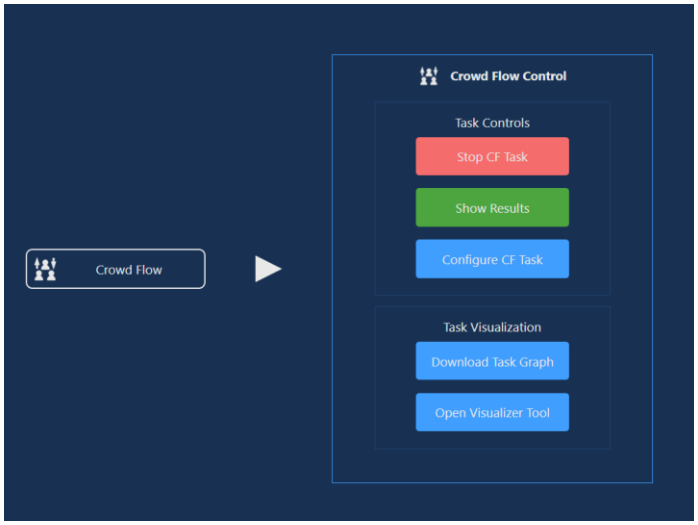

Crowd Flow
Crowd Flow Control
Task Controls
Stop CF Task
Show Results
Configure CF Task
Task Visualization
Download Task Graph
Open Visualizer Tool

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.

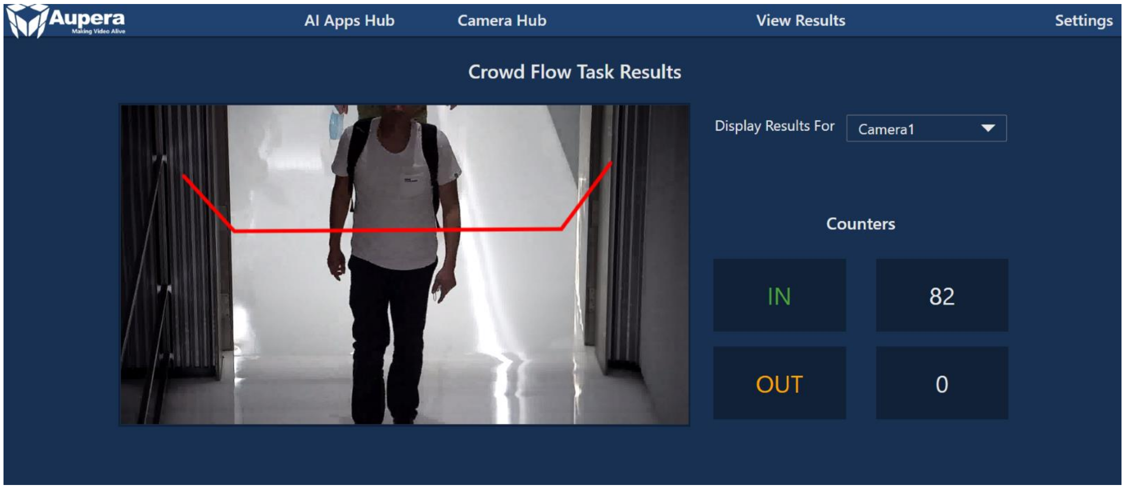

Aupera
Making Video Alive
AI Apps Hub
Camera Hub
View Results
Settings
Crowd Flow Task Results
Display Results For
Camera1
Counters
IN
82
OUT
0

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****.*

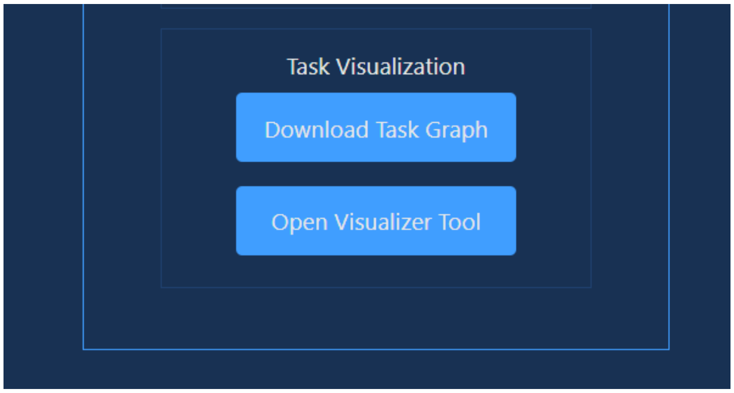

Task Visualization
Download Task Graph
Open Visualizer Tool

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). 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.

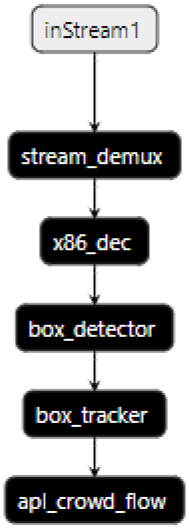

inStream1
stream_demux
x86_dec
box_detector
box_tracker
apl_crowd_flow

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

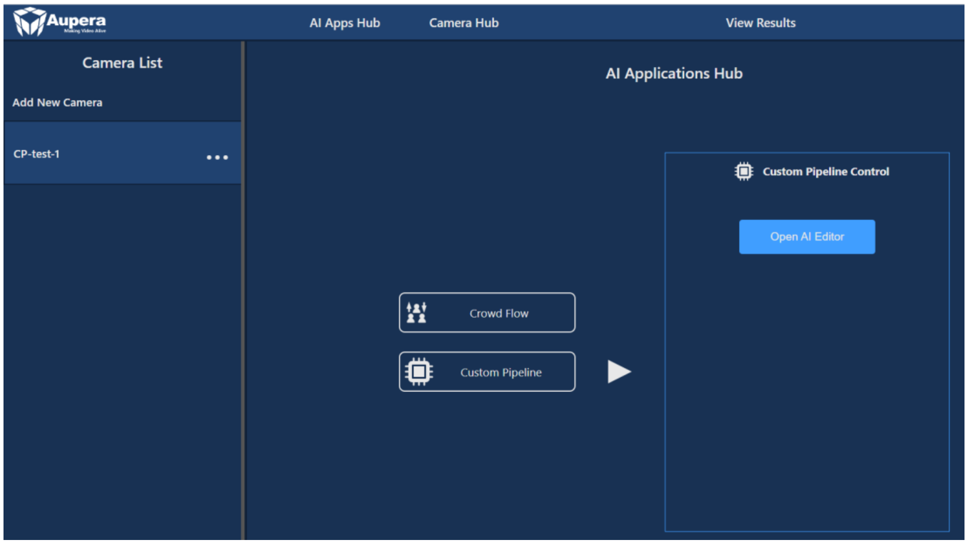
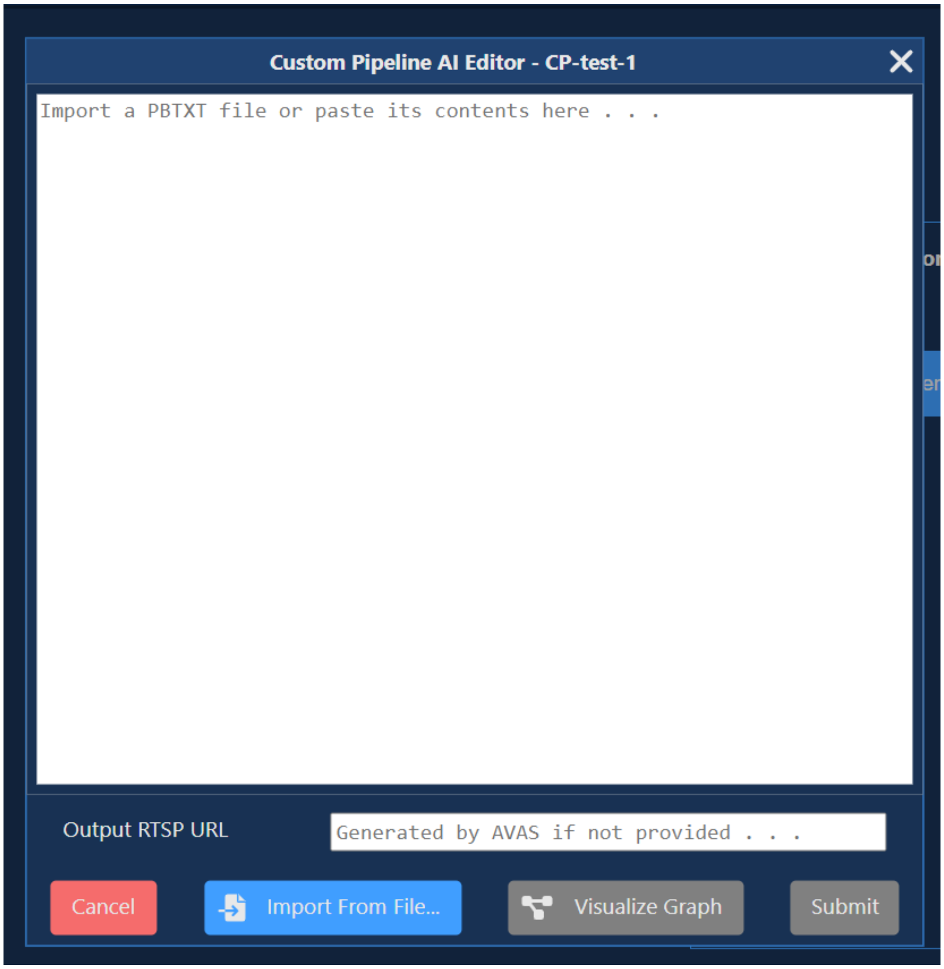

Aupera
AI Apps Hub
Camera Hub
View Results
Camera List
Add New Camera
CP-test-1
AI Applications Hub
Custom Pipeline Control
Open AI Editor
Crowd Flow
Custom Pipeline

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). If you're running Custom Pipeline through Aupera Web Client, please only use the files that include "using_rtsp" in their name.



Custom Pipeline AI Editor - CP-test-1

Import a PBTXT file or paste its contents here . . .

Output RTSP URL: Generated by AVAS if not provided . . .

Cancel
Import From File...
Visualize Graph
Submit

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.

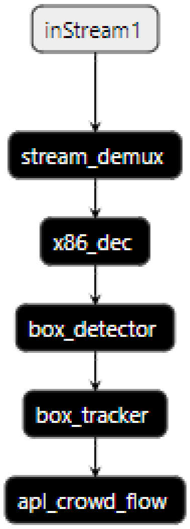
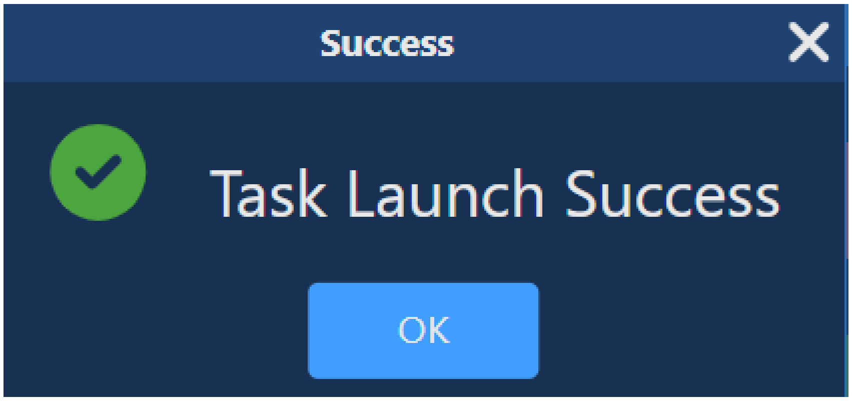

inStream1
stream_demux
x86_dec
box_detector
box_tracker
apl_crowd_flow

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.

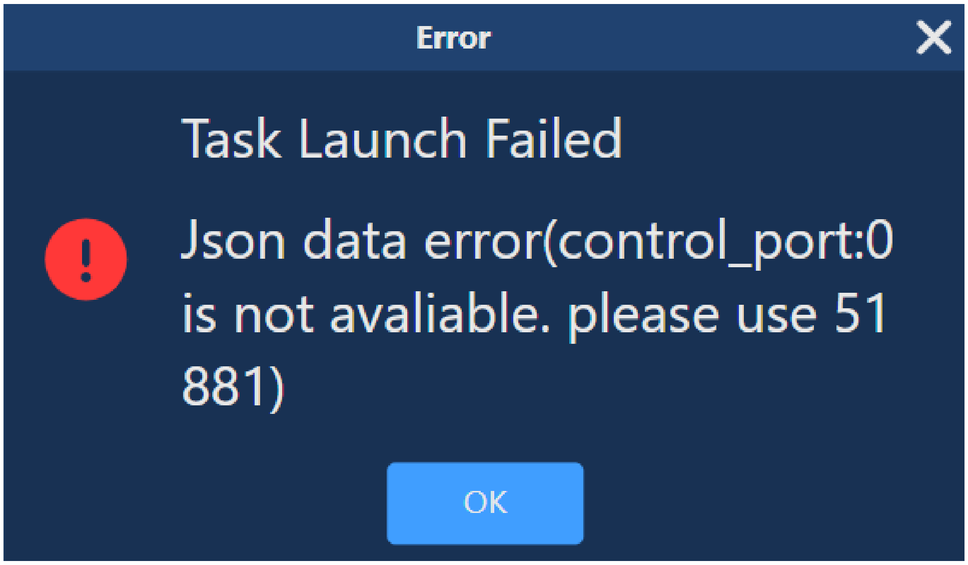
NOTE: **<user_specifcied_name>** can be any arbitrary name that the user chooses.

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



Task Launch Success

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

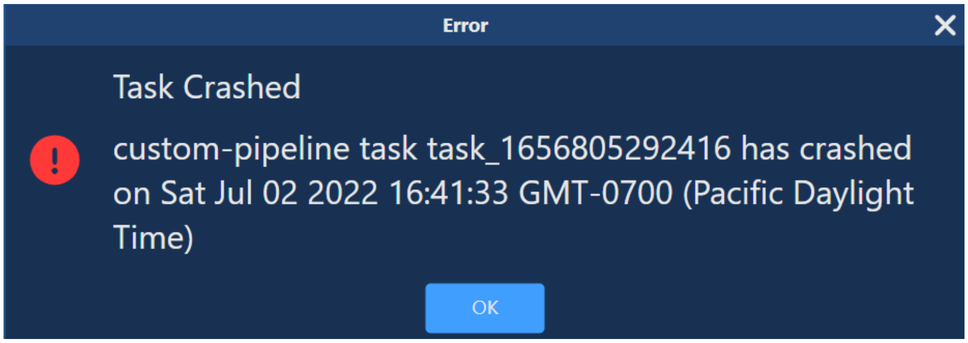


Task Launch Failed

Json data error(control_port:0 is not avaliable. please use 51881)

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



Task Crashed

custom-pipeline task task_1656805292416 has crashed on Sat Jul 02 2022 16:41:33 GMT-0700 (Pacific Daylight Time)

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.

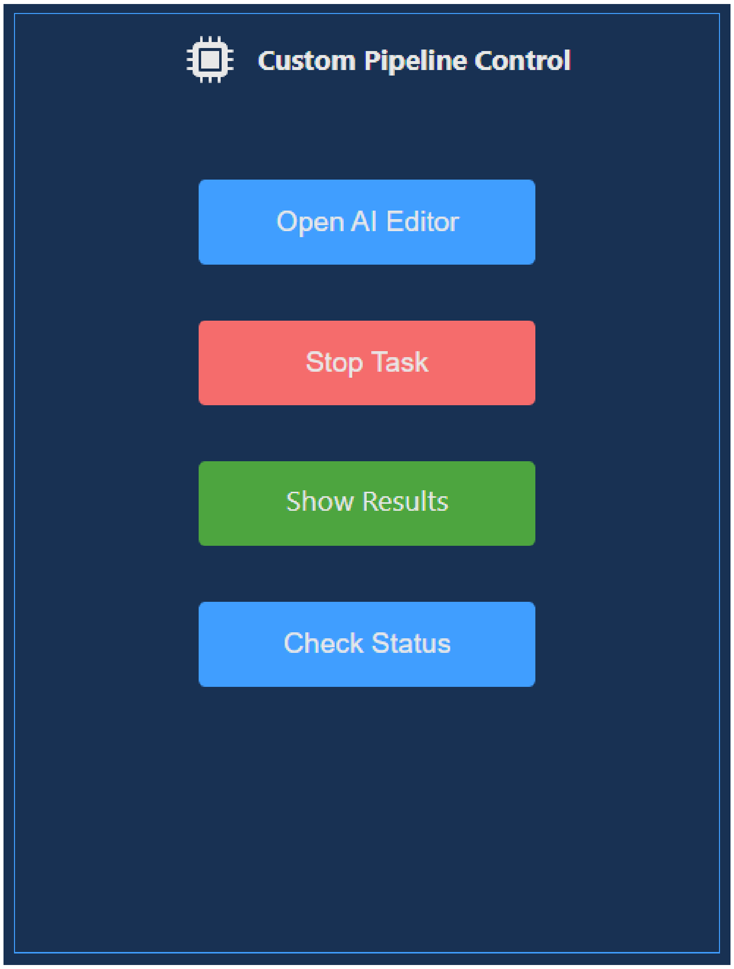
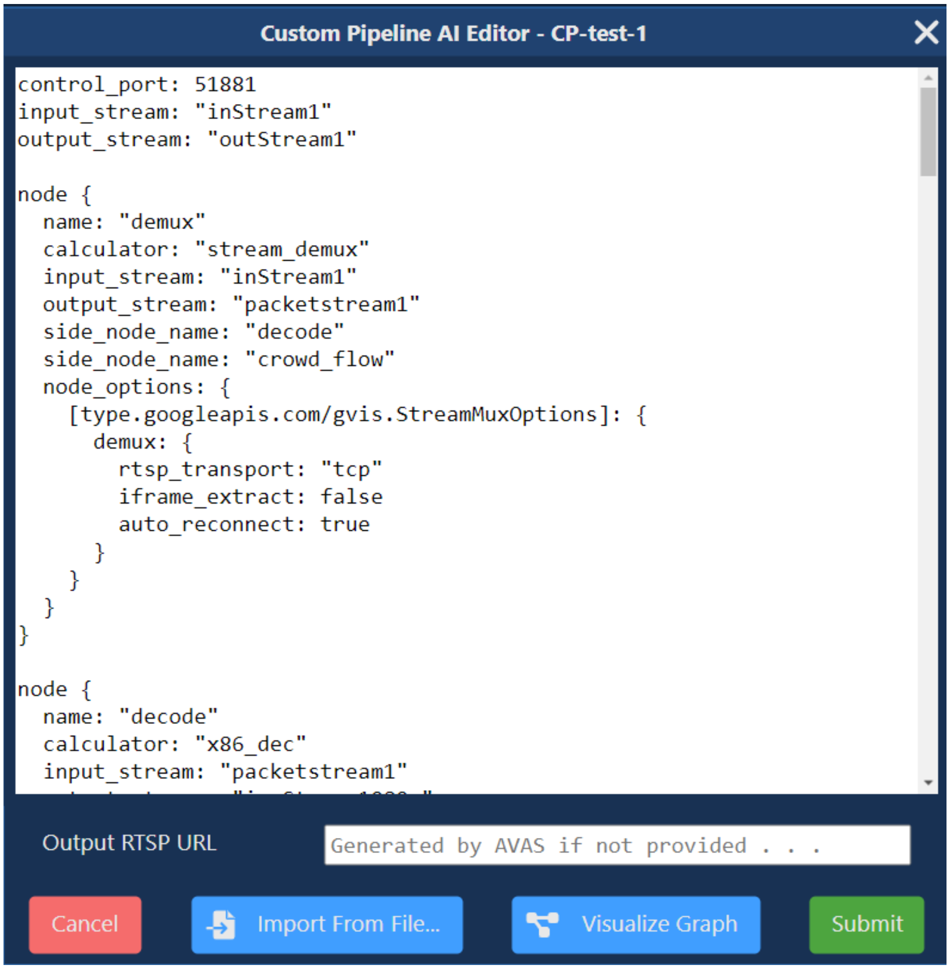

Custom Pipeline Control
Open AI Editor
Stop Task
Show Results
Check Status

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.


Custom Pipeline AI Editor - CP-test-1
Output RTSP URL
Generated by AVAS if not provided . . .
Cancel
Import From File...
Visualize Graph
Submit

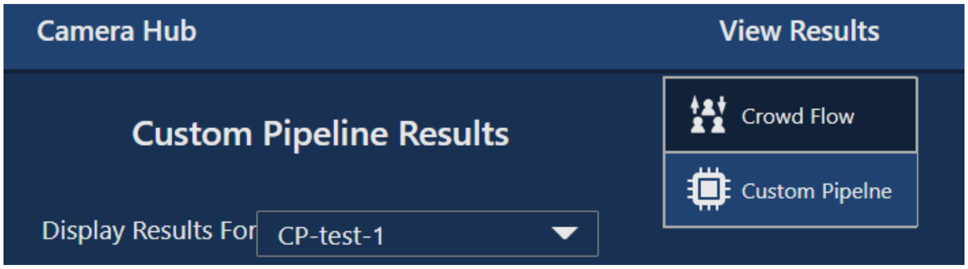
```
control_port: 51881
input_stream: "inStream1"
output_stream: "outStream1"

node {
  name: "demux"
  calculator: "stream_demux"
  input_stream: "inStream1"
  output_stream: "packetstream1"
  side_node_name: "decode"
  side_node_name: "crowd_flow"
  node_options: {
    [type.googleapis.com/gvis.StreamMuxOptions]: {
      demux: {
        rtsp_transport: "tcp"
        iframe_extract: false
        auto_reconnect: true
      }
    }
  }
}

node {
  name: "decode"
  calculator: "x86_dec"
  input_stream: "packetstream1"
```

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

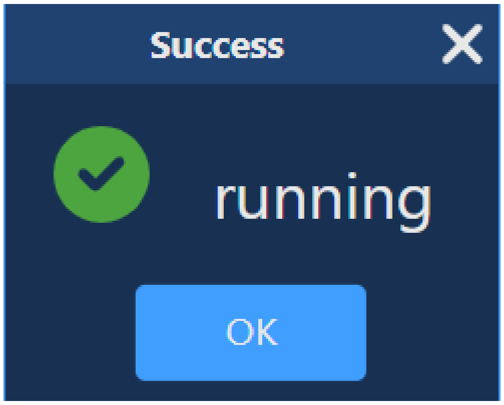
I. To see the CP task results (output video), either click on ***Show Results*** in the CP control component or navigate to CP Results using the Header, then choose desired camera in the ***Display Results For*** list.


Camera Hub
View Results
Custom Pipeline Results
Crowd Flow
Custom Pipelne

Display Results For CP-test-1

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.



running

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

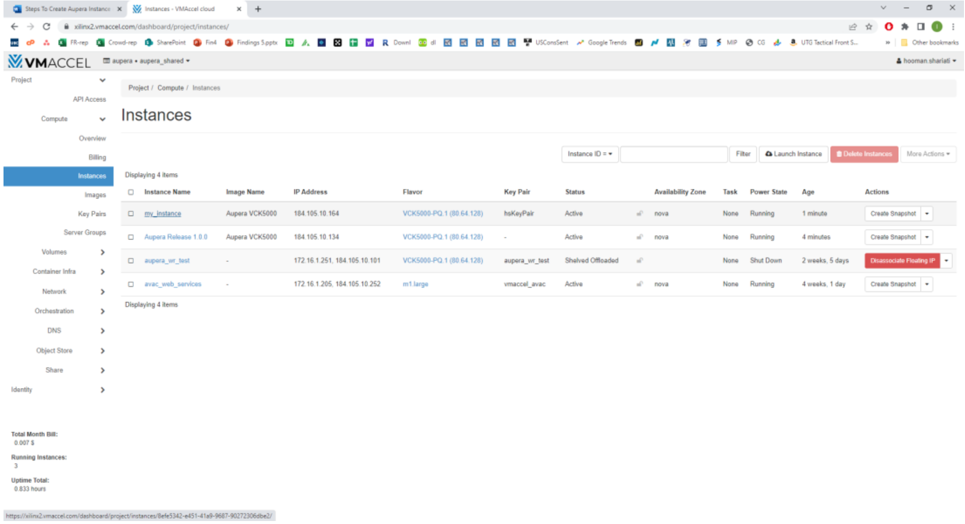
K. To stop the task, click Stop Task in the CP control component.

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.


VMACCEL
aupera • aupera_shared
Project
API Access
Compute
Overview
Billing
Instances
Images
Key Pairs
Server Groups
Volumes
Container Infra
Network
Orchestration
DNS
Object Store
Share
Identity
Project / Compute / Instances
Instance ID =
Filter
Launch Instance
Delete Instances
More Actions

Instances

Displaying 4 items

☐	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
☐	my_instance	Aupera VCK5000	184.105.10.164	VCK5000-PQ.1 (80.64.128)	hsKeyPair	Active	nova	None	Running	1 minute	Create Snapshot
☐	Aupera Release 1.0.0	Aupera VCK5000	184.105.10.134	VCK5000-PQ.1 (80.64.128)	-	Active	nova	None	Running	4 minutes	Create Snapshot
☐	aupera_wr_test	-	172.16.1.251, 184.105.10.101	VCK5000-PQ.1 (80.64.128)	aupera_wr_test	Shelved Offloaded		None	Shut Down	2 weeks, 5 days	Disassociate Floating IP
☐	avac_web_services	-	172.16.1.205, 184.105.10.252	m1.large	vmaccel_avac	Active	nova	None	Running	4 weeks, 1 day	Create Snapshot

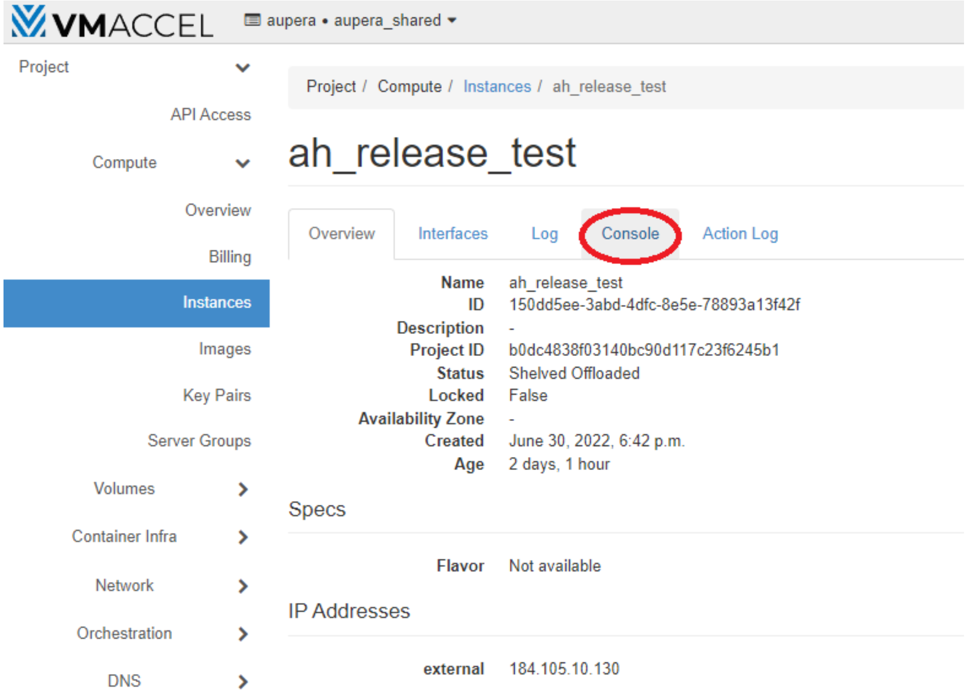
Displaying 4 items

Total Month Bill:
0.007 $
Running Instances:
3
Uptime Total:
0.833 hours

https://xilinx2.vmaccel.com/dashboard/project/instances/8efe5342-e451-41a9-9687-90272306dbe2/

Figure 2.27. VMAccel instances page – instance selection

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


VMACCEL
aupera • aupera_shared
Project
API Access
Compute
Overview
Billing
Instances
Images
Key Pairs
Server Groups
Volumes
Container Infra
Network
Orchestration
DNS
Project / Compute / Instances / ah_release_test
Overview
Interfaces
Log
Console
Action Log

ah_release_test

Name ah_release_test
ID 150dd5ee-3abd-4dfc-8e5e-78893a13f42f
Description -
Project ID b0dc4838f03140bc90d117c23f6245b1
Status Shelved Offloaded
Locked False
Availability Zone -
Created June 30, 2022, 6:42 p.m.
Age 2 days, 1 hour

Specs

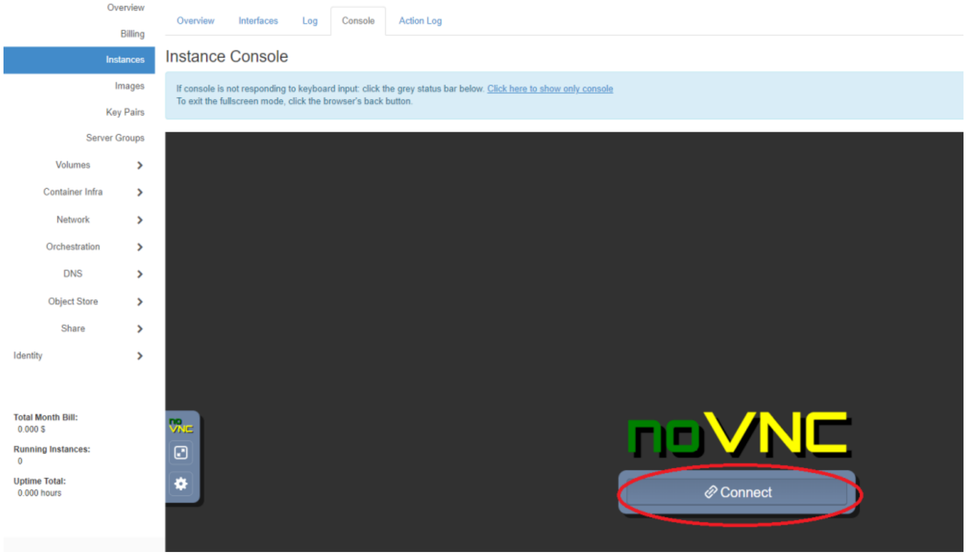
Flavor Not available

IP Addresses

external 184.105.10.130

Figure 2.28. VMAccel instances page – instance console

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

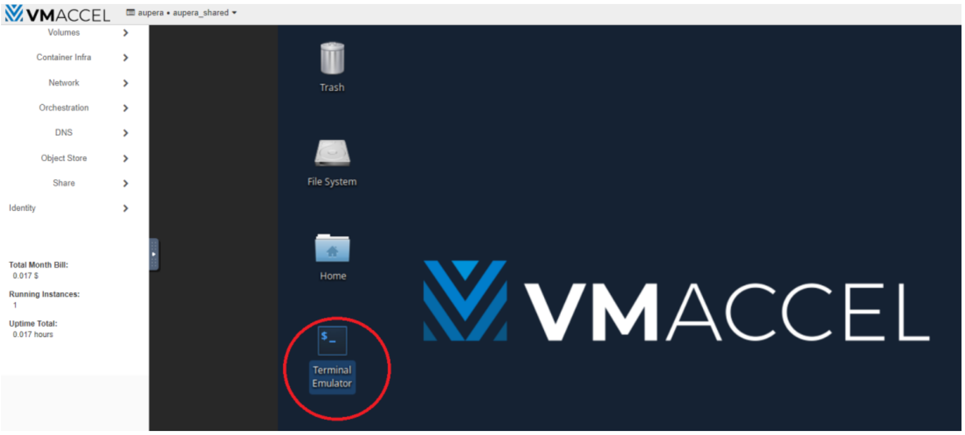

Overview
Billing
Instances
Images
Key Pairs
Server Groups
Volumes
Container Infra
Network
Orchestration
DNS
Object Store
Share
Identity
Overview
Interfaces
Log
Console
Action Log
Instance Console
noVNC
Connect

If console is not responding to keyboard input: click the grey status bar below. Click here to show only console
To exit the fullscreen mode, click the browser's back button.

Total Month Bill:
0.000 $
Running Instances:
0
Uptime Total:
0.000 hours

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.

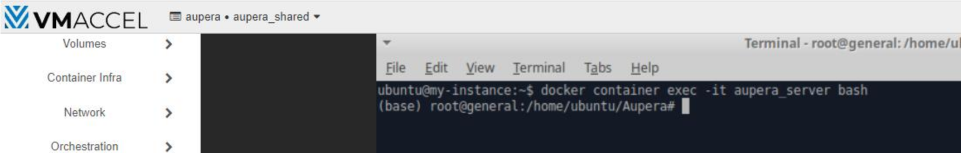

VMACCEL
aupera • aupera_shared
Volumes
Container Infra
Network
Orchestration
DNS
Object Store
Share
Identity
Trash
File System
Home
Terminal Emulator
VMACCEL

Total Month Bill:
0.017 $
Running Instances:
1
Uptime Total:
0.017 hours

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
```


VMACCEL
aupera • aupera_shared
Volumes
Container Infra
Network
Orchestration
Terminal - root@general: /home/u
File
Edit
View
Terminal
Tabs
Help

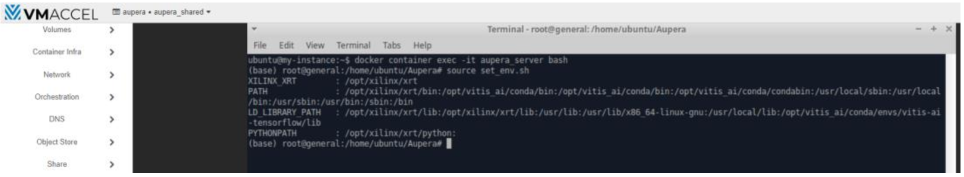
```
ubuntu@my-instance:~$ docker container exec -it aupera_server bash
(base) root@general:/home/ubuntu/Aupera# 
```

ex

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
```



```
ubuntu@my-instance:~$ docker container exec -it aupera_server bash
(base) root@general:/home/ubuntu/Aupera# source set_env.sh
XILINX_XRT        : /opt/xilinx/xrt
PATH              : /opt/xilinx/xrt/bin:/opt/vitis_ai/conda/bin:/opt/vitis_ai/conda/bin:/opt/vitis_ai/conda/condabin:/usr/local/sbin:/usr/local
/bin:/usr/sbin:/usr/bin:/sbin:/bin
LD_LIBRARY_PATH   : /opt/xilinx/xrt/lib:/opt/xilinx/xrt/lib:/usr/lib:/usr/lib/x86_64-linux-gnu:/usr/local/lib:/opt/vitis_ai/conda/envs/vitis-ai
-tensorflow/lib
PYTHONPATH        : /opt/xilinx/xrt/python:
(base) root@general:/home/ubuntu/Aupera# 
```

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.

NOTE: Usually, you can kill the VMSS2.0 tasks (launched via the command line) by pressing CTL + C on your keyboard. For the throughput measurements tasks, you might need to press CTL + C a few times and / or use CTL + | combination to kill all the threads.

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 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:

A. 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.

B. 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.

3 RUNNING VMSS2.0 ON-PREMISES

3.1 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
```

3.2 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_VCK500
0-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)

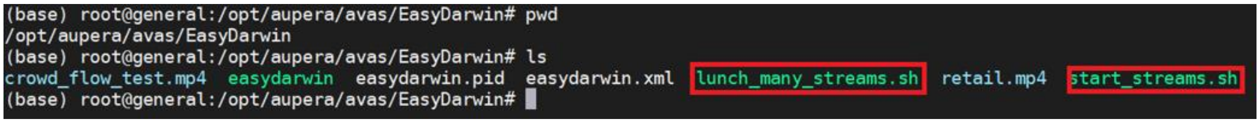
H. If you are not running the start.sh script (i.e., you're planning on just using the command line to run pipelines without using the client/ avas server), then you need to source the xbutil and Vitis environments by running the command:

```
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.

3.3 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:



```
(base) root@general:/opt/aupera/avas/EasyDarwin# pwd
/opt/aupera/avas/EasyDarwin
(base) root@general:/opt/aupera/avas/EasyDarwin# ls
crowd_flow_test.mp4  easydarwin  easydarwin.pid  easydarwin.xml  lunch_many_streams.sh  retail.mp4  start_streams.sh
(base) root@general:/opt/aupera/avas/EasyDarwin# 
```

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

A. 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)

B. 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
```

4 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. 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:

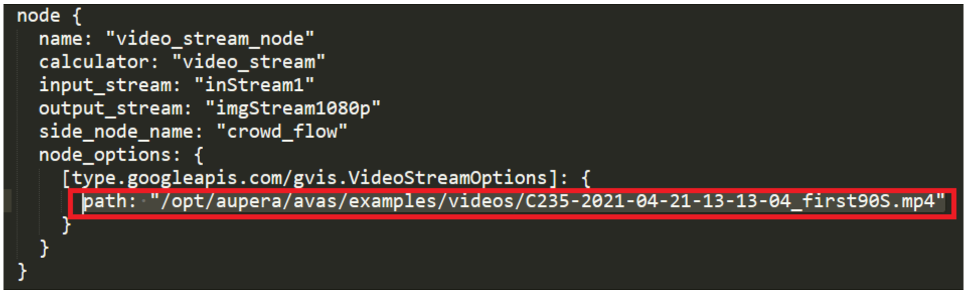
```
avaser -i input.pbtxt -o output.pbtxt -c using_rtsp.pbtxt
```

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 than what we have provided, you will need to modify the "path" parameter in the video_stream node. As shown in Figure 4.1 below:



```
node {
  name: "video_stream_node"
  calculator: "video_stream"
  input_stream: "inStream1"
  output_stream: "imgStream1080p"
  side_node_name: "crowd_flow"
  node_options: {
    [type.googleapis.com/gvis.VideoStreamOptions]: {
      path: "/opt/aupera/avas/examples/videos/C235-2021-04-21-13-13-04_first90S.mp4"
    }
  }
}
```

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.

NOTE 4: Usually, you can kill the VMSS2.0 tasks (launched via the command line) by pressing CTL + C on your keyboard. For the throughput measurements tasks, you might need to press CTL + C a few times and / or use CTL + | combination to kill all the threads.

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

A. box_detector/using_rtsp_0output.pbtxt

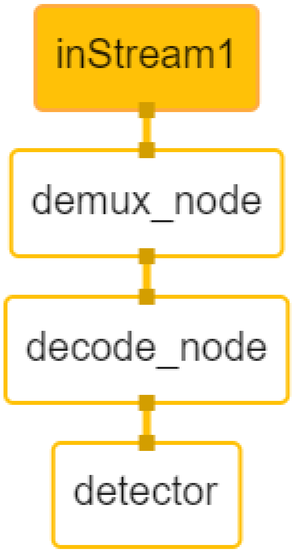

inStream1
demux_node
decode_node
detector

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

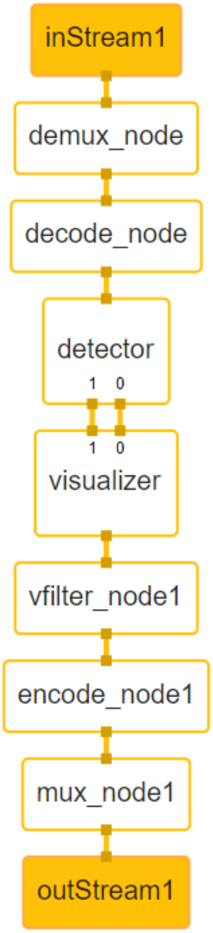

inStream1
demux_node
decode_node
detector
1 0
1 0
visualizer
vfilter_node1
encode_node1
mux_node1
outStream1

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 :

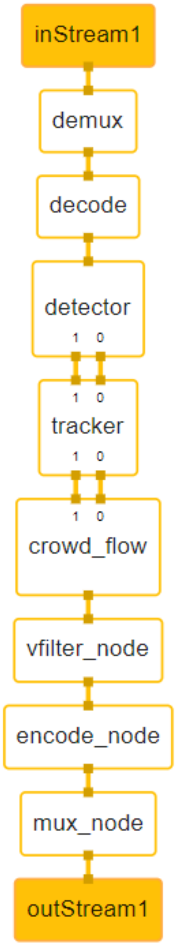

inStream1
demux
decode
detector
1 0
1 0
tracker
1 0
1 0
crowd_flow
vfilter_node
encode_node
mux_node
outStream1

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

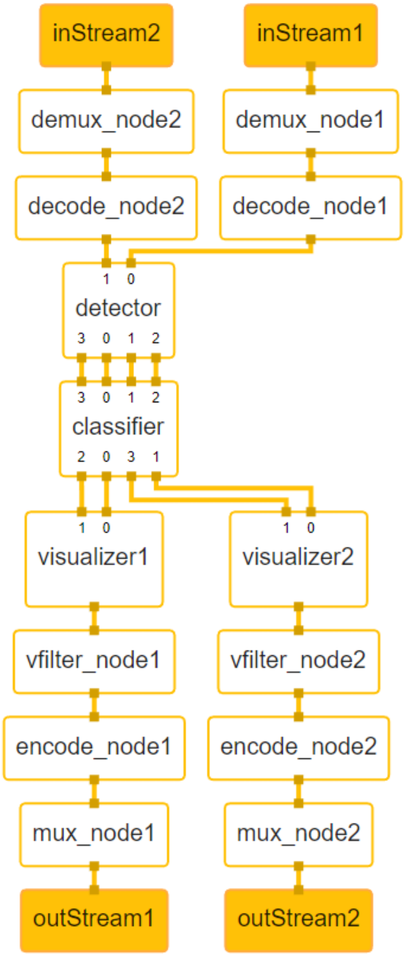

inStream2
inStream1
demux_node2
demux_node1
decode_node2
decode_node1
1 0
detector
3 0 1 2
3 0 1 2
classifier
2 0 3 1
1 0
visualizer1
1 0
visualizer2
vfilter_node1
vfilter_node2
encode_node1
encode_node2
mux_node1
mux_node2
outStream1
outStream2

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

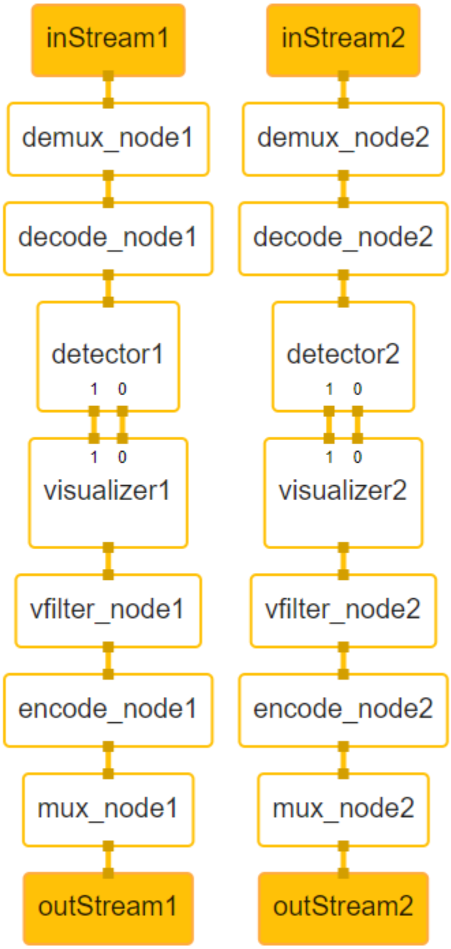

inStream1
inStream2
demux_node1
demux_node2
decode_node1
decode_node2
detector1
1 0
detector2
1 0
1 0
visualizer1
1 0
visualizer2
vfilter_node1
vfilter_node2
encode_node1
encode_node2
mux_node1
mux_node2
outStream1
outStream2

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.

5 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.

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) 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 PacketPtr<Detections> objects.

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 PacketPtr<Detections> stream. The order of the output streams will be the same as the input streams.9

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

PacketPtr<Detections> objects. In this way, the frames from different input streams will be merged into a single PacketPtr<Detections> output stream (i.e., neither the frames nor the streams will be in order).

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 one output stream is specified, the first one will always be of the type PacketPtr<Tracks> containing the detected (and predicted) bounding boxes.

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 PacketPtr<Classifications> objects.
- 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 false, 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 PacketPtr<Classifications> to the output streams. There is a one-to-one relationship between the input frames streams and output streams

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 PacketPtr<Classifications> stream. The order of the output streams will be the same as the input streams.

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 is similar to decode node and outputs a stream of GvisVframes. The images are iterateed in alphabetical order. After the last image is reached, the 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 it is best to use some arbitrary dummy rtsp stream that does not exist.

This node has a single output that similar to the 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;
```

6 CREATING VMSS2.0 PIPELINES

VMSS 2.0 pipelines can be created using the collection of nodes provided by Aupera's node toolkit or using custom nodes. For instructions on how to create your own nodes please refer to the next section (section 5) of this document. The following is a list of the most important nodes currently provided in the toolkit; however, we are constantly adding new nodes (so this list might not be comprehensive).

6.1 Major Operations Supported at Graph Level

Currently The VMSS2.0 graph supports three main operations:

1. Initializing the Graph

```
StatusCode initialize(std::vector<std::string> input_urls,
std::vector<std::string> output_urls);
```

This operation validates the graph to ensure that:

1. There is a URL provided for every input stream and every output stream declared at the graph level

2. At the node level, every input stream has either a graph level input stream (matched to a URL) or a node level output stream (i.e., is connected to an output of another node).

3. All the nodes that the graph uses have been registered with the framework

Once the conditions above have been satisfied, the start method of every node is called to complete the graph initialization.

2. Running the Graph

```
StatusCode run();
```

This method calculates how many threads are required for each node that provides an execute method (nodes that don't provide an execute method indicate to the framework that they're responsible for their own thread management).

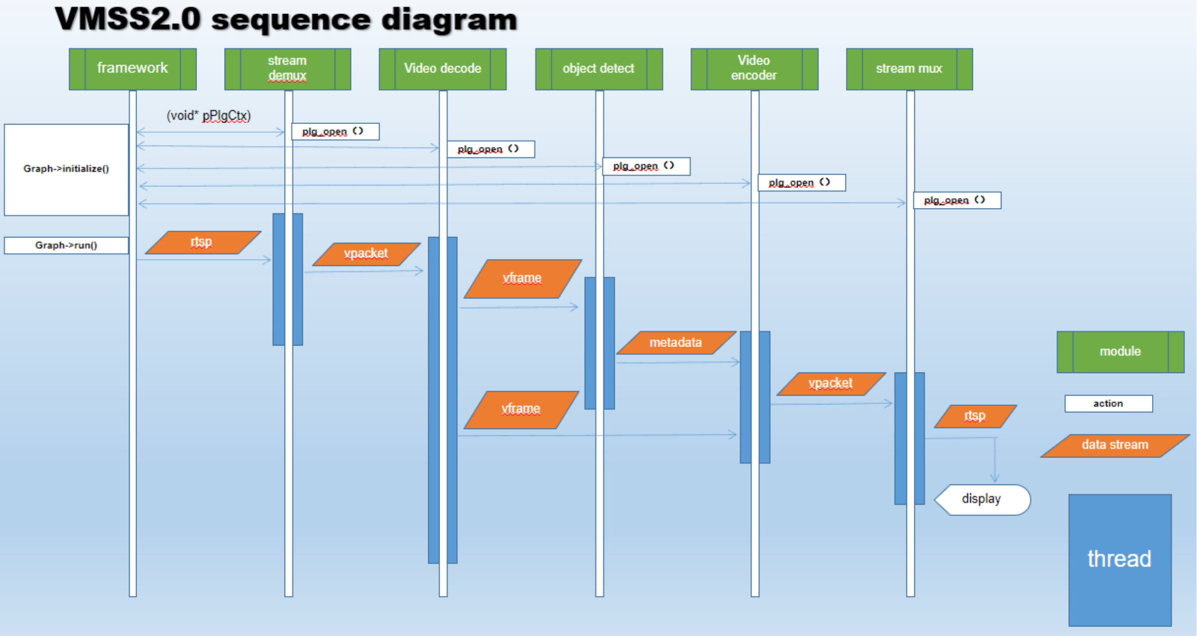
By default, each node (with an execute method) will receive its own thread. However, this behavior can be modified both at the node level, where certain nodes specify that they require more than one thread; or at the graph level, where the total number of threads specified for the graph is more than the total number of nodes in the thread. In the latter case, the framework will determine the exact number of nodes each thread receives based on the node level configurations.

3. Destroying a Graph

```
StatusCode uninitialize();
```

Once there are no more input packets received from the input streams, or an error occurs, the framework will call the stop method of every node to give them an opportunity to clean up their data and exit gracefully.

The diagram below shows the sequence of actions described above that VMSS2.0 framework takes at graph level.


VMSS2.0 sequence diagram
framework
stream demux
Video decode
object detect
Video encoder
stream mux
(void* pPlgCtx)
plg_open ()
plg_open ()
plg_open ()
plg_open ()
plg_open ()
Graph->initialize()
Graph->run()
rtsp
vpacket
vframe
metadata
vframe
vpacket
rtsp
display
module
action
data stream
thread

6.2 Synchronization and Ordering

The logic by which the framework calls the execute method of a given node can be specified at a node level. Users can provide their own custom synchronization logic by using synchronization nodes. However, the framework provides several synchronization behaviors on nodes that provide an execute method (no synchronization available on nodes that opt for manual data management by not providing an execute method). To request synchronization on any node that uses an execute method, you need to include the **stream_sync.drop_strategy** tag in your node's configuration (in the pbtxt file defining the pipeline). Optionally, a node can include the **stream_sync.timeout_ms** to specify how long the framework should wait for the packets from the different input streams to become available (to perform the synchronization successfully). By default, this is set to 12 seconds.

If the **stream_sync.drop_strategy** tag is not included or is specified as **“DROP_INCOMPLETE_PACKETS”** for node, node will drop the packets if there are missing packets. However if it is defined as “**NEVER_DROP**”, it would always feed the packets to even to the node even if incomplete.

In order to opt-out of synchronous inputs, the input stream types should be set to “**UNSYNCED”** in which case packets will be available for the node as soon as they are available.

The framework will buffer the packets from the earlier streams while waiting for the delayed streams. An incremental value such as Presentation Time Stamp (PTS) or Generation Time Stamp (GTS), or Frame Number is used to maintain the ordering between packets. Any incremental value can be used by the user (even custom values) by adding it to the enqueue methods of the previous node(s). By default, the framework uses PTS.

Finally, it should be noted that the framework will always maintain the ordering provided by previous nodes. For example, since the video decoder node will output the decoded frames in order, the frames will be passed in-order to all subsequent nodes unless one of the nodes explicitly changes the ordering.

6.3 Side-Packet Communications

Side packet communication is also achieved through streams. Side packet stream sinks are marked with **type** attribute of **SIDE_PACKET**. In which case the sink can always read the most recent packet

In the example below the object detector node sends the parameter detect_interval to all the nodes that it has a side packet relationship with during its initialization phase (before its starts running or receives any packets):

```
Node->enqueue(side_packet_output_stream_index,
make_packet<DetectIntervalPacket>(detect_interval));
```

In the example below the object tracker node receives the detect_interval parameter from the object detector node during its initialization (before its execute method is called or it receives any packets). This node will wait for the message “detect interval” for a certain period of time (up to DETECT_INTERVAL_WAIT_TIMEOUT) and will move on without it if it hasn't received the message during that time.

```
PacketPtr<DetectIntervalPacket> detect_interval_packet;
auto error_code = node->dequeue_block(
    side_packet_input_stream_index, detect_interval_packet);
if (error_code != EC_OK) {
    // handle error case
}
```

6.4 Input/Output Streams

The graph can receive a list of input streams and a list of output streams. There are only two rules regarding these (graph level) inputs and outputs:

A. The order of input/output stream names must be the same order by which the corresponding URLs have been communicated to the server.

B. Every graph-level input/output stream must be connected (i.e., be the input or output of) exactly one node.

Each node can also receive a list of input streams and a list of output streams. There are only two rules regarding these (node level) inputs and outputs as well:

A. The names of the inputs and outputs must be unique in the graph. You must have the same name once as an input and at least once as an output.

B. Each input must be connected to either a graph level input stream or a node level output stream. Each output must either be connected to a graph level output stream or to one or more node level input streams with the same name.

For both graph and node level inputs and outputs, you can also use the “name:tag” format to add a tag for display purposes.

In the case that a stream connection should go backwards, (create a loop) is should be marked as UNSYNCED.

The designers of VMSS2.0 pipelines are not limited by a particular hierarchy of nodes; therefore, you can have back-edges in the exact same way as normal edges. For example, Node1 can have an input stream A that is connected to output stream A of Node2. At the same time, Node1 can also have an output stream B that is connected to input stream B of Node2. Please note that synchronization cannot be performed on back edges.

6.5 Data streams

VMSS2.0 supports arbitrary data streams between nodes. In other words, there are no limits on the type of data or the number of data streams that nodes can pass to each other. All types of streams are optimized for memory usage by ensuring that the data never has to be deep copied when being passed from one node to another. All data types (whether defined by the framework or defined by different nodes) must extend the **PacketBase** class shown below:

```
class PacketBase
{
public:
    inline const std::chrono::system_clock::time_point&
get_gen_time_point() const;
    inline const timestamp_t get_sync_timestamp() const;
    inline size_t get_itr_no() const;
    inline const bool is_continued() const;
};
```

Note that the framework can communicate any class that extends the PacketBase class above across different nodes. Additionally, the framework will handle all reference counting and control. In other words, the user does not need to worry about allocating or deallocating memory buffers.

Furthermore, the framework will take care of all meta data conversions. So the user does not need to worry about performing any packet/metadata conversion. Metadata corresponding to ROIs (detected bounding boxes) will be always returned in a normalized format (floating point between 0 to 1) with respect to the coordinate system of the original frame.

Finally, the framework allows the user to access their data in OpenCv types whenever possible since majority of users (especially the ones focused on Computer Vision) are familiar with these data types.

Below, you can see the fields contained in the Image class which the framework defines for all frames and images communicated across different nodes.

```
class ImagePacket : public PacketBase {
public:
    // get cv::Mat
    cv::Mat& get_cv_mat() const;
    // get pixel format. Can be RGB, BGR, etc
    PixFmtType get_pix_fmt() const;
    // get dimentions
    void get_dims(cv::Size&) const;
    void get_dims(int& height, int& width) const;
    // get dimensions of original image
    void get_orig_dims(cv::Size&) const;
    void get_orig_dims(int& height, int& width) const;
    // get ROI on original image
    void get_roi_on_orig(cv::Rect2f&) const;
};
```

The rest of this section will include some of the meta data defined by individual nodes (included in Aupera's node toolkit) to provide an example for how different nodes can define their own metadata to communicate between nodes.

6.6 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) 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
```

6.7 Node Level Configurations

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

```
    /*
     * this denotes the calculator name
     * this can be an arbitrary string determined by graph
author in order to
     * uniquely identify the node by.
     */
    string name;

    /*
     * the vendor of the calculator. This determines who/what
company wrote the
     * node this is useful to differentiate same calculators via
different
     * vendors
     */
    string vendor;

    /* in avaf, it means processor plugin name. Plugin name is
known as
     * calculator */
    string calculator;

    /* node input stream name, can be multiple, each stream name
should be
     * unique for current graph
     */
    repeated string input_stream;

    enum ContractNoYes {
      CONTRACT = 0;
      NO = 1;
      YES = 2;
    }

    message InputStreamAttributes {
      string name;  // input stream name referring to
input_stream field
      /*
       * InputStreamAttributes:
       * name: name of input stream
       * CONTRACT: stream type to be fetched from contract
       *
       * SYNCED_IMMUTABLE: stream type is forward. Forward
streams are not
       * allowed to contain loops and they are synced. IMMUTABLE
denodes that
       * the node is not allowed to modify these packets via
sync-time-stamp.
       * the node is not allowed to modify this packet
       *
       * SYNCED_MUTABLE: stream type is forward. The different
is that this kind
       * of sink allows the node to modify the packet
       *
       * UNSYNCED_IMMUTABLE: is a type of stream that can have
any direction
       * whether forwards or backwards. The packets of these
streams are always
       * immutable meaning that the sink is not allowed to
modify it.
       *
       * SIDE_PACKET: these streams can have any direction and
not synced and
       * timestamped like UNSYNCED_IMMUTABLE. The difference is
that these
       * streams only keep track of the latest packet as opposed
```

```
to UNSYNCED
       * which act as a queue. These streams are useful for
communicating
       * various configurations
       */
      enum Type {
        CONTRACT = 0;
        SYNCED_IMMUTABLE = 1;
        SYNCED_MUTABLE = 2;
        UNSYNCED_IMMUTABLE = 3;
        SIDE_PACKET = 4;
      }
      Type type;
      /*
       * This denodes if the sink expects continueable packets.
This means
       * consecutive packets with the same timestamp
       */
      ContractNoYes can_continue;
    }

    // list of attributes for all the input streams
    repeated InputStreamAttributes input_stream_attributes;

    // node output stream name, can be multiple, each stream
should be unique
    // for current graph
    repeated string output_stream;

    message OutputStreamAttributes {
      string name;  // name of input stream
      /*
       * This denotes the capacity of the stream. In the case
that it is not
       * defined, global graph capacity will be used for this
specific stream
       * instead
       */
      uint32 capacity;
      /*
```

```
       * This determines the behavior of
aup::avaf::Node::enqueue member
       * function in the case that stream's full capacity is
reached
       *
       * BLOCK: block the enqueue function call until room is
available in the
       * stream via dequeueing from other nodes
       *
       * DROP_FRONT: automatically drop the oldest packet (front
of the stream)
       * to make room for the new packet being enqueued
       *
       * FAIL: fail the enqueue call immediately in case that
stream is full
       */
      enum OnFullAct {
        BLOCK = 0;
        DROP_FRONT = 1;
        FAIL = 2;
      }
      OnFullAct on_full_act;
      /*
       * same as can_continue in input_stream_attributes. Node
that if source is
       * continuable, all the sinks must be continueable. The
reverse is not
       * true so non-continueable source may be connected to
continueable sinks.
       */
      ContractNoYes can_continue;
    }

    repeated OutputStreamAttributes output_stream_attributes =
7;

    message StreamSync {
      /*
       * DropStrategy determines the scenario that framework
drop the packet in
```

```
       * case that timeout is reached
       *
       * CONTRACT: determient the packet dropping strategy from
the contract
       *
       * DROP_INCOMPLETE_PACKETS: drop if any packets are
missing or incomplete.
       * incomplete packets apply to continueable streams which
have some
       * packets for the same timestamp but not all of them
       *
       * DROP_MISSING_PACKETS: drop if there are any packets for
any of the
       * streams are completely missing
       *
       * NEVER_DROP: never drop packets. if timeout is reached
and one synced
       * input is available
       */
      enum DropStrategy {
        CONTRACT = 0;
        DROP_INCOMPLETE_PACKETS = 1;
        DROP_MISSING_PACKETS = 2;
        NEVER_DROP = 3;
      }
      /*
       * ReuseStrategy: determines how framework should replace
the missing
       * packets in case that
       *
       * EMPTY: dont reuse last packets. let them be empty
       *
       * REUSE_LAST_COMPLETE_PACKETS: reuse last packets only if
they are
       * complete
       *
       * REUSE_LAST_AVAILABLE_PACKETS: reuse any available last
packet
       */
      enum ReuseStrategy {
        CONTRACT2 = 0;
        EMPTY = 1;
        REUSE_LAST_COMPLETE_PACKETS = 2;
        REUST_LAST_AVAILABLE_PACKETS = 3;
      }
      enum SyncTimeStampMatch {
        CONTRACT3 = 0;  // sync based on what is available in
the contract
        EQUAL = 1;      // sync based on sync-time-stamp
equality
      }
      DropStrategy drop_strategy;
      ReuseStrategy reuse_strategy;
      SyncTimeStampMatch sync_match;
      int32 timeout_ms;  //  timeout which is calculated based
on generation time
    }

    StreamSync stream_sync;

    // the private options passing to each node, defined by
users
    repeated google.protobuf.Any node_options;
```

7 CREATING VMSS2.0 NODES

7.1 Steps for creating VMSS2.0 nodes

To create your own VMSS2.0 node you need to perform the 6 steps below:

1. Write a .proto file (according to protobuf 3 protocol) to specify the options that your node could receive from the .pbtxt file specifying a pipeline which includes your node. For example, the following specifies the options for the object (box) visualizer node (included in Auepra's node toolkit and explained in section 5.3.9):

```
syntax = "proto3";

package aup.avaf;

message BoxVisualizerOptions {
    message Color {
        float r = 1;
        float g = 2;
        float b = 3;
    }

    message ClassColor {
        Color color = 1;
        int32 label = 2;
    }

    message Offset {
        float x = 1;
        float y = 2;
    }
    enum InputType {
        INPUT_TYPE_DETECTION = 0;
        INPUT_TYPE_CLASSIFICATION = 1;
    }
    InputType input_type = 1;
    Color text_color = 2;
    Color box_color = 3;
    repeated ClassColor class_color = 4;
    Color default_class_color = 5;
    Offset text_offset = 6;
    int32 box_thickness = 7;
    int32 text_size = 8;
    int32 font = 9;
    double font_scale = 10;
    int32 font_thickness = 11;
    int32 line_type = 12;
}
```

2. Create the class defining the behavior of your node. All VMSS2.0 nodes must extend the **CalculatorBase** class. Note that CalculatorBase is a template class and will require the name of the protobuf class specifying the options for your node. In our example, this will be the **BoxVisualizerOptions** class that we created in step 1. In the code snippet below, you can see the header file (box_visualizer.hpp) that declares the BoxVisualizerCalculator class which specifies the behavior of our box visualizer node.

```
class BoxVisualizerCalculator : public
CalculatorBase<BoxVisualizerOptions>
{
     unordered_map<int, Scalar> class_colors;
     Scalar default_class_color;
     int init_no;
     thread viz_thread;
     void viz_worker();
     void handle_detections();
     void handle_classifications();

protected:
     void generate_contract() override;

public:
     BoxVisualizerCalculator(const ::aup::avaf::Node* node) :
CalculatorBase(node) {}
     ~BoxVisualizerCalculator();
     bool initialize() override;
};
```

3. Register your new node in the framework. To do this you have to call one of the two registration macros provided by the framework. Normally you would use the macro AUP_AVAF_REGISTER_CALCULATOR(vendor_, name_, class_, options_, description_, hw_type_masks_...)

A. vendor: Name of the vendor of the calculator

B. name: Name of the calculator which normally is in snake case

C. class: Calculator class name

D. options: Options of the calculator which are defined via the .proto file

E. description: Description is an optional description for the calculator. Can be left as an empty string

F. hw_type_masks: is the list of supported hard types which can be left empty as {}. If filled, framework will validate if the calculator is runnable on the current hardware stack or not

Below, you can see the macro call to register our box visualizer node:

```
AUP_AVAF_REGISTER_CALCULATOR(“Aupera”, “box_visualizer”,
             aup::avaf::BoxVisualizerCalculator,
             aup::avaf::BoxVisualizerOptions,
             "box visualizer calculator",
             {aup::avaf:: HARDWARE_TYPE_X86_64,
              aup::avaf:: HARDWARE_TYPE_X86}
)
```

In case that you don’t want to use framework features (for advanced users only) you can use the macro AUP_AVAF_REGISTER_CALCULATOR_EXT which gives the ability to not have execute callback. Using this registry is an advanced topic is out of the scope of this document.

4. Provide an **initialize** method for your node. This method will be called right after your node is constructed. To write an **initialize** method, you need to override the one provided by the **CalculatorBase** class in the declaration of your class. Like below:

```
Aup::avaf::ErrorCode  intialize(std::string& err_str) override;
```

You will also need to define the **initialize** method in the definition of your class. For details on how to write execute methods read section 7.2.

5. Now you will need to provide an execute method for your node. To do this you need to override the execute() method provided by the CalculatorBase class in the class declaration of your node. Like below:

```
ErrorCode execute() override;
```

You will also need to define the execute method in the definition of your class. For details on how to write execute methods read section 7.3.

6. Optionally, you can provide a contract for the inputs to your node. If you do so, then the framework will perform the following configuration and validations:

- Validate the types of source and sinks of each stream are compatible
- Enforce if the stream sink types are compatible with eacho ther.
- Mark other behaviors choices for the stream inputs outputs
- Select synchronization schemes

Most of these contract parameters are useful when you intend your node to be used by some other third party. If you decide to register a contract for your node, then in the declaration of the class for your node, you will need to override the f**ill_contract()** method as shown in the example above.

Here is an example for fill_contract of license-plate recognizer node:

```
ErrorCode PlateRecognizerCalculator::fill_contract(shared_ptr <
Contract > & contract)
{
  if (contract -> input_stream_names.size() != 2) {
    return EC_SIZE_MISMATCH;
  }
  if (contract -> output_stream_names.size() > 2) {
    return EC_SIZE_MISMATCH;
  }
  vector < aup::detect::Detector::DetectedObject > res;
  cv::Mat empty;
  contract -> sample_input_packets[0] = make_packet < Detections
> (0, 0, 0, res, empty, 0);
  contract -> sample_input_packets[1] = make_packet <
ImagePacket > ();
  if (contract -> output_stream_names.size() == 0) {
    contract -> input_attrs_arr[1].set_type(contract ->
input_attrs_arr[1].SYNCED_MUTABLE);
  }
  if (contract -> output_stream_names.size() > 0) {
    contract -> sample_output_packets[0] = make_packet <
PlateRecognitions > (0);
  }
  if (contract -> output_stream_names.size() > 1) {
    contract -> sample_output_packets[1] = make_packet <
ImagePacket > ();
  }
  contract -> stream_sync.set_drop_strategy(contract ->
stream_sync.DROP_INCOMPLETE_PACKETS);
  return EC_OK;
}
```

7.4 Node Initialization

As you saw in above description, the open method of the node will call the initialize method of each node class. The job of the node specific initialization method is to validate the provided input parameters, log them for the user, send or receive any required side packets, and proceed to initializing the member variables that the node requires.

Most of the nodes in Aupera’s toolkit use libraries contained in our collection of “Hyper Plugin Specific Libraries”. In the example below, you can see the initialization of a rudimentary box detector node.

```
ErrorCode BoxDetectorCalculator::initialize(string& err_str){
    BOOST_LOG_SEV(pNode->logger()->mlogger(), LOG_INFO) <<
    "\033[33m" << __func__ << " box detector options:" <<
    "\033[0m" << endl;

    BOOST_LOG_SEV(pNode->logger()->mlogger(), LOG_INFO) <<
    "\033[33m" << __func__ << " detector type = " <<
    options->detector_type() << "\033[0m" << endl;
    BOOST_LOG_SEV(pNode->logger()->mlogger(), LOG_INFO) <<
    "\033[33m" << __func__ << " kernel name = " <<
    options->kernel_name() << "\033[0m" << endl;


    aup::detect::Detector::InitParams init_params = {};
    init_params.detector_type = options->detector_type();
    init_params.kernel_name = options->kernel_name();
    detector = aup::detect::Detector::create(init_params);

    //send the detect_interval as a side message to next node
    node->enqueue(side_packet_index,

make_packet<DetectIntervalPacket>(detect_interval));

    return EC_OK;
}
```

As you can see above, this method will parse the options passed to the node and logs them for the user (this is something all nodes should do as a styling standartd). Next, it initializes Aupera's detector library. Finally, it base64 encodes the detect_interval parameter (that it has received from the user) and queues it as a side packet for downstream nodes.

7.5 Defining the execute method

NOTE: if you decide to use an execute method you will need to

1. execute

```
ErrorCode execute() override;
```

2. Impelmented the execute method for your class as described below.
3. Use the IMPL_CALCULATOR_WITH_EXECUTE macro to register your node.

The job of the execute method is to receive the inputs from the framework, perform the requested operations, and return the requested outputs. In most cases, the structure of execute methods look very similar to the example below.

```
ErrorCode AplCrowdFlowCalculator::execute()
{
  // check the status of the graph (i.e pipeline)
  if (node->get_graph_status() != GRAPH_STATUS_RUNNING) {
    BOOST_LOG_SEV(node->logger()->mlogger(), LOG_INFO)
    << __func__ << " called with graph status of: "
    << node->get_graph_status() << endl;

    return EC_ERROR;
  }

  auto& streamsData = node->get_stream_data();

  //read the 1st input stream (error if it doesn't exist)
  auto data_from_1s_stream =
    dynamic_packet_cast<Tracks>(streamsData.at(0));

  //read the 2nd input stream (which may or may not exist)
  if (streamsData.size() > 1) {
    auto images =
dynamic_packet_cast<Images>(streamsData.at(1));
    cv::Mat cvFrame = images->getMat();
  }

  //do some work to produce some “result”

  //if there are output streams attached
  //send some results to the 1st output stream
  if (node->outputStreamsName.size() > 0) {
    auto errorCode = node->enqueue(0, result);
    if (errorCode) {
      BOOST_LOG_SEV(node->logger()->mlogger(), LOG_ERROR)
      << " vframe enqueue to: " <<
      node->outputStreamsName.at(0).c_str()
      << " failed. ErrorCode: " << errorCode << endl;

      return EC_ERROR;
    }
  }
  return EC_OK;
}
```

In the example above, we first make sure that the graph is in running status.

Then we read the 1st input (from the first input stream). In this example, we are assuming that the first input stream is mandatory (i.e., the node doesn’t first check whether there is an input steam), so an error is thrown if this node has no input streams.

Then, we check if there is a second input stream. If so, then we will read the 1st packet available on this stream. The node assumes that this packet is of type “Images” containing a frame. The next line retrieves the data contained in the images object into cv::Mat format to perform some operation.

Finally, we check whether there is an output stream available to this node. If so, we enqueue our result object for the 1st output stream of this node.

NOTE: The results in this example can be any class that extends the **PacketBase** class. Since any such class can be transmitted across different nodes.