Developing a Prototype Mask Detecting Smart Camera Using the NVIDIA Jetson Nano: A Hands-On Evaluation

By the Staff and Partners of

March 2021

Overview

This report presents an independent, hands-on evaluation of using the NVIDIA Jetson Nano, an embedded processor designed for vision and AI/ML, to prototype a real-world edge AI application: MaskCam, a smart camera capable of estimating the number and percentage of people wearing face masks in its field of view. The evaluation was led by BDTI, an independent technology analysis firm, working closely with its partners, Tryolabs S.A. and Jabil Optics.

Our evaluation sought to answer questions such as: How difficult is it to create an application using the Jetson Nano andNVIDIA's tools and SDKs? How complete is NVIDIA's support ecosystem, including documentation and community presence? How much effort is required to integrate the Jetson Nano with off-the-shelf hardware available through NVIDIA's hardware partners? Where did things work well, and where did we encounter snags or sharp edges?

We present our developer journey, from product concept to prototype. We describe the requirements and specifications for MaskCam, the hardware and software architecture we chose, and the steps required to implement them on both the Jetson Nano Developer Kit and an off-the-shelf carrier board with a Jetson Nano system-on-module (SOM). Working with Jabil we provide cost estimates for manufacturing MaskCam in volume.

Overall, our team was impressed with the Jetson Nano and its ecosystem. The quality documentation and examples for the NVIDIA SDKs, the breadth of the hardware partners and modules available to be used with the Jetson Nano, and the containerization tools from balena all facilitated rapid development of our smart mask detection camera. The fact that we were able to conceptualize, design, and create a production-ready prototype of MaskCam in a short time with a small team speaks to this. While there were certainly challenges, we were able to overcome all of them.

If you are interested in MaskCam, we encourage you to try it out: the MaskCam source code is available under the MIT License at https://github.com/bdtinc/maskcam. If you have a Jetson Nano Developer Kit and a USB web camera, you can get the MaskCam software running on your system with two simple commands described in the README. Please feel free to email maskcam@bdti.com if you have questions.

1. Introduction

This report presents BDTI's independent, hands-on evaluation of using the NVIDIA Jetson Nano to prototype a real-world edge AI application: a smart camera capable of estimating the percentage of people wearing face masks in its field of view.

For the evaluation, BDTI and its partners created a mask detection smart camera based around the Jetson Nano and worked through the development process from concept to production prototype. We leveraged the tools, SDKs, and hardware resources available from the NVIDIA Jetson ecosystem and recorded our experiences throughout the developer journey.

This evaluation seeks to provide:

- A real-world example of using the Jetson Nano in an edge-based machine learning application, from concept to (almost) production
- A sense of the effort and learning curve required to create a smart camera product using the Jetson Nano
- A qualitative analysis of our experiences with the Jetson Nano and its ecosystem

The target audience for this report is product developers, including engineers, engineering managers, and product managers, interested in understanding the experience of developing a Jetson Nano-based product with computer vision and machine learning functionality.

The application code and design details for the project have been open-sourced under the MIT License on GitHub and are available at https://github.com/bdtinc/maskcam.

2. About BDTI, Tryolabs, and Jabil Optics

This independent evaluation was led by Berkeley Design Technology, Inc. (BDTI), a technology analysis and software development firm specializing in embedded computer vision and deep learning applications. BDTI has extensive experience developing, optimizing and deploying computer vision applications across many different platforms. In addition to its software development work, for almost 30 years BDTI has performed in-depth, hands-on evaluations of numerous processors, development kits and tools. For more information about BDTI, please visit https://www.bdti.com/. For questions about this report, please email us at info@bdti.com.

In this evaluation BDTI worked closely with two partners, Tryolabs S.A. and Jabil Optics:

Tryolabs is a machine learning consulting company that focuses on transforming businesses by building and deploying AI-powered solutions. With more than a decade of experience in the field, Tryolabs has helped over 110 companies transform and embrace solutions based on leveraging data and applying state-of-the-art AI technology. The Tryolabs team is composed of experts in applying computer vision, natural language processing, and predictive analytics techniques. From the discovery of opportunities, R&D to actual production implementation,

Tryolabs helps clients increase revenue, reduce costs and generate a competitive advantage using AI. For more information about Tryolabs, please visit https://tryolabs.com.

Jabil Optics is recognized worldwide as a leader in advanced optical design and manufacturing. With 170 employees across four locations, Jabil Optics provides advanced optical design, process development, supply chain management and precision manufacturing services to realize its customers' product goals. Jabil Optics experience is built on decades of solving complex optical problems for our customers in the automotive, consumer electronics, healthcare and industrial markets. Jabil Optics is part of the larger Jabil Inc., a worldwide manufacturing concern with 100 plants in 30 countries. For more information about Jabil Optics, please visit https://www.jabil.com/capabilities/optics.html.

This project was funded by NVIDIA but was executed independently. NVIDIA personnel received periodic briefings on our progress and in some cases offered suggestions for approaches to try. NVIDIA personnel reviewed a pre-publication draft of this report and were given the opportunity to offer factual corrections. That said, ultimately, all technical decisions in this project were made by BDTI and its partners, and BDTI made the final decisions on what went into this report.

3. The NVIDIA Jetson Nano and its Ecosystem

The NVIDIA Jetson family comprises both developer kits and system-on-modules (SOMs) designed for edge AI applications and devices where power and space are limited. Table 1 summarizes the Jetson family [1]. It includes the 128 CUDA-core Nano, the 256 CUDA-core TX2 series, the 512 CUDA-core AGX Xavier, and the 384 CUDA-core Xavier NX. The Jetson processors include hardware-based accelerators for speeding up various parts of the AI pipeline: a GPU for accelerating inference and vision, a programmable vision accelerator, a video image compositor, and video codecs for accelerating multimedia. NVIDIA provides access to these accelerators via user level libraries for end-to-end AI acceleration.

The Jetson Nano is the lowest cost family member and features four Arm Cortex-A57 cores and a GPU based on NVIDIA's Maxwell architecture. The Jetson Nano Developer Kit [2] combines the Nano Module with an NVIDIA-built carrier board featuring a wide range of interfaces, including MIPI, USB, HDMI, ethernet, and GPIO. The Jetson Nano module is pin-compatible with the Xavier NX and the just-released TX2 NX Modules, providing an upgrade/downgrade path depending on budget and performance requirements.

	Nano Series		TX2 Series				Xavier NX	AGX Xavier
	Nano 2 GB	Nano 4 GB	TX2 4 GB	TX2	TX2i	TX2 NX		
GPU	128-core NVIDIA Maxwell GPU		256-core NVIDIA Pascal GPU				384-core NVIDIA Volta GPU with 48 Tensor Cores	512-core NVIDIA Volta GPU with 64 Tensor Cores
CPU	Quad-core Arm Cortex-A57 MPCore processor		Dual-core Denver 2 64-bit CPU and quad-core Arm Cortex-A57 MPCore processor				6-core NVIDIA Carmel ARMv8.2 64-bit CPU 6 MB L2 4 MB L3	8-core NVIDIA Carmel Armv8.2 64-bit CPU 8 MB L2 4 MB L3
Memory	2 GB 64-bit LPDDR4 25.6 GB/s	4 GB 64-bit LPDDR4 25.6 GB/s	4 GB 128-bit LPDDR4 51.2 GB/s	8 GB 128-bit LPDDR4 59.7 GB/s	8 GB 128-bit LPDDR4 (ECC Support) 51.2 GB/s	4 GB 128-bit LPDDR4 51.2 GB/s	8 GB 128-bit LPDDR4x 51.2 GB/s	32 GB 256-bit LPDDR4x 136.5 GB/s
Power	5 W, 10 W		7.5 W, 15 W		10 W, 20 W	7.5 W, 15 W	10 W, 15 W	10 W, 15 W, 30 W
Video Decode (H.265)	1x 4K60 2x 4K30 4x 1080p60 8x 1080p30		2x 4K60 4x 4K30 7x 1080p60 14x 1080p30				2x 4K60 4x 4K30 12x 1080p60 16x 1080p30	2x 8K30 6x 4K60 26x 1080p60 72x 1080p30
DL Accelerator	n/a						2x NVDLA Engines	
Vision Accelerator	n/a						7-Way VLIW Vision Processor	
Connector	n/a	69.6 mm x 45 mm 260-pin SO-DIMM connector	87 mm x 50 mm 400-pin connector			69.6 mm x 45 mm 260-pin SO-DIMM connector	69.6 mm x 45 mm 260-pin SO-DIMM connector	100 mm x 87 mm 699-pin connector
Dev Kit Cost	$59	$90	No longer available (end of life)			See Xavier NX Dev Kit	$397	$699
Module Cost	n/a	$129 qty 1 $99 qty 1K	$299 qty 1	$479 qty 1	$849 qty 1	$199 qty 1	$479 qty 1	$999 qty 1

Table 1: NVIDIA Jetson family members and characteristics.

The lowest cost version of the Jetson Nano is the Nano 2 Gbyte Developer Kit. Aimed at hobbyists and educational users, it is available for $59, but not available as a SOM. The 4 Gbyte Jetson Nano Developer Kit used in this project is priced at $90, well below the rest of the Jetson family, and is available as both a developer kit and SOM.

Note that NVIDIA's Developer Kits are not intended to be used for production. Rather, NVIDIA expects product makers to purchase SOMs (which they call "modules") for production use. These modules must be used with a carrier board, which may be either custom designed or available off the shelf from an ecosystem partner. NVIDIA does not sell individual Jetson processor chips, meaning that Jetson Modules are the only path available for volume manufacturing of Jetson-based products. The Jetson Nano module used in this project is $129 in unit quantities, and $99 in quantities of 1,000.

The default software environment for the Jetson family is JetPack SDK [3]. JetPack is based on Ubuntu 18.04 Linux and includes the NVIDIA Jetson Linux Driver Package (L4T), a CUDA-X

software stack that includes NVIDIA developer tools, and the DeepStream SDK framework used in this project.

The Jetson family is supported by a rich ecosystem of dozens of hardware, software, sensor, and developer tools partners.

4. Our Application: A Mask Detecting Smart Camera

The COVID-19 pandemic has created a need for understanding the number of individuals wearing or not wearing face masks in public areas. Our application, MaskCam, addresses this need by detecting and counting masked and unmasked individuals in a broad area of coverage, and reporting statistics on the overall ratio of mask wearers to a remote web server.

MaskCam is intended to help both public and private organizations understand mask compliance. For example, a local transportation authority might use MaskCam to understand the percentage of people wearing masks on a train station platform at rush hour. Similarly, it can help a store manager see the week-over-week increase in mask wearers after they implement a "masks required" policy. Ultimately, MaskCam allows business owners, public officials, and other users to be better informed in their efforts to limit the spread of COVID-19.

In an effort to address the demands of the application, we identified both top-level and detailed requirements, and from those, we created our product specifications.

Top-Level Requirements:

- Mask Detection: Determine percentage of mask wearers in an area from a surveillance camera perspective
    - Count and track the number of people in a large viewing area (90°+ horizontal field of view, at up to 35 feet from the camera)
    - Determine whether individuals are masked or unmasked
    - Calculate the overall ratio of masked people to unmasked people
- Informatics: Report statistics on number and percentage of mask wearers over time to the cloud; the resulting statistics will be viewable via web browser
- Streaming: Stream camera view with detection results and live statistics over IP
- Video storage: Optionally store video for playback purposes on a customer-supplied SD card
    - Records processed video (showing person and mask detections) to on-device storage medium
    - Indicates periods of interest in the video (e.g., unusually large number of people or very low mask-wearing percentage)
    - Maximum length of recorded video depends on size of SD card
- Environments: Able to work in a variety of visual and physical environments
    - Indoors and outdoors
    - Daytime (and nighttime with sufficient external lighting)

    - High-angle perspectives and face-on perspectives
- Hardware: Based on the Jetson Nano module, but may upgrade to other Jetson modules if more compute is needed
- Cost: Total BOM and manufacturing cost of approximately $300

Product Specifications

Based on the above requirements, we created the specifications shown in Table 2.

Item	Specification
Processor and memory	NVIDIA Jetson Nano SOM (4x Arm Cortex A57, 128-core Maxwell GPU, 4 Gbytes RAM, 16 Gbytes eMMC)
WiFi	802.11b/g/n
LTE	Optional via mini-PCIe card (since some installation environments may have neither wired Ethernet nor WiFi)
Power budget	15 W (10 W to Jetson Nano, 5 W to peripheral components)
External power	Supplied via barrel connector or POE (since some environments have POE and some don't)
Wired Ethernet	1000 Mbit/sec
SD card	Optional, removable SD card for video storage
Audio	None
Camera resolution	4K @ 30 FPS for video (inference can be performed at 5-10 FPS)
Camera field of view	90 degrees or greater

Table 2: MaskCam product specifications.

The choice of image sensor resolution is worth a few words of explanation. Because we want MaskCam to work in a variety of visual environments, and to be able to detect masks at a distance, as well as to provide headroom for additional pre- or post-processing we specified a 4K image sensor. While a lower resolution sensor would have been suitable for the inference being performed, we determined that the cost savings didn't justify the reduction in capability.

5. Evaluation Methodology

BDTI began with the product concept for MaskCam and approached the development process in a typical fashion: we created specifications to define the product's functionality and features, and then worked on getting a "lab prototype" working with the Jetson Nano Developer Kit. We then developed a rapid time-to-market version using off-the-shelf hardware components. Finally, we researched options for designing a cost-down high volume product.

Overall, we divided the project into three phases:

In phase 1, we decided on requirements and specifications and designed the product, exploring the NVIDIA ecosystem for relevant Jetson hardware, software, and services, and set cost and performance targets. In particular, we:

- Decided on requirements and created specifications, including performance and cost targets
- Researched NVIDIA offerings, the NVIDIA ecosystem, and open source to identify relevant resources, e.g., cameras, boards, tools, algorithms, software components and frameworks, and services
- Evaluated the most promising resources to understand their capabilities and limitations; and then selected resources
- Identified an off-the-shelf mask detection algorithm and dataset to be used
- Designed the hardware/software architecture of the system, considering interfaces, capabilities, and limitations of chosen resources

In phase 2, we implemented core product functionality for the mask detector using relevant Jetson tools. In particular, we:

- Obtained the chosen resources (algorithm, test data, boards, cameras, tools, frameworks, software components, etc.) and gained familiarity with them
- Brought up and characterized each component
- Created the full system, integrating components and optimizing where possible
- Verified functionality, debugging as needed
- Measured performance and briefly optimized to meet performance specifications

In phase 3, we defined a path to volume production that meets the cost target. In particular we:

- Engaged with a hardware design and manufacturing partner
- In consultation with the hardware partner, determined the extent of custom hardware design required to meet the cost target
- Obtained detailed proposal and cost quote for custom hardware design, manufacturing NRE, and volume production for quantities of 10K and 100K units per year

Our team comprised:

- A project lead (25% time)
- A hardware lead (50% time)
- An embedded software and ML lead (100% time)

- A containerization software engineer (50% time for two months)
- A cloud software engineer (100% time for two weeks)

We worked on the project for roughly four months (November 2020 - February 2021, with a break for the holidays). During this time we were able to go from concept to working prototype, with a clear path to production.

In the process, we recorded our experiences working with the Jetson Nano and its ecosystem throughout development. We focused on topics such as:

- How difficult is it to create an application for the Jetson Nano using NVIDIA's tools and SDKs?
- How complete is NVIDIA's support ecosystem, including documentation and community presence?
- How much effort is required to integrate the Jetson Nano with off-the-shelf hardware available through NVIDIA's hardware partners?
- Where did things work well, and where did we encounter snags or sharp edges?

6. Software Development

Initial Implementation

MaskCam's initial face mask detection algorithm was based on an existing algorithm developed by Tryolabs [4]. This algorithm was written in Python and designed to run on desktop GPUs. It used a two-stage inference process: it performed pose estimation to locate faces and then applied image classification to classify whether detected faces were masked or unmasked. It also used a Tryolabs-developed open source object tracker called Norfair to follow detections across the scene. For each person that walks past the camera, Norfair tracks their face detection box as it moves across the scene. It only counts the individual once, rather than counting them in every frame. Then, if the detection score for their face is above a fixed threshold for several frames, a voting system decides if the person is wearing a mask or not, or if the face is not visible enough to tell. This process allows us to leverage information from the whole video sequence instead of just individual frames. The final output of this algorithm is a count of the number of individuals who have passed by the camera and the percentage that were wearing a mask.

This algorithm, while accurate, required too much compute to run on the Jetson Nano as written. As part of moving to the Nano, we simplified the algorithm to use a single-shot object detection model, recognizing the following object classes:

- Face with mask
- Face without mask
- Face not visible
- Misplaced mask

The *face not visible* class is needed for tracking purposes, since we need to follow heads even if they are not looking at the camera at times.

As there was no previous object detection dataset with these categories, we manually curated a dataset of mask training images by downloading images from the Internet and also extracting frames from downloaded videos. We labeled the images using CVAT, an open-source image labeling tool. This dataset contains 351 training images with 6,843 annotations across all classes, and that's what we used during most of the development process. After the development process was largely finished, we expanded this dataset by combining it with other publicly available sources, as described at the end of this section.

Model Selection

We evaluated three different models as candidates for our use case. The main tradeoff on each individual model was the input resolution vs. inference time. We tested MobileNetV2+SSD, YOLOv4-full, and YOLOv4-tiny [5] at different input resolutions and batch sizes. We converted the models to TensorRT, ran them on the Jetson Nano, and measured the engine inference time, excluding any pre-processing (like resizing frames) or post-processing (like non-maxima suppression). The results are shown in Table 3. The inference FPS is the pipeline bottleneck, since we know we can't improve it by parallelizing any other processing tasks.

Model	Input Resolution	Batch Size	Jetson Nano FPS (Inference)
MobileNetV2+SSD	300x300	1	25
		8	35
	1024x608	Any	6
YOLOv4-full	608x608	Any	2.5
YOLOv4-tiny	608x608	Any	21
	1024x608	Any	14

Table 3: Inference FPS on Jetson Nano for examined neural network models and input resolutions.

One result that was clear after these experiments was that the batch size only made a difference at small input resolutions on the Jetson Nano. For larger input resolutions, batching frames to perform inference did not reduce runtime compared to processing frames individually.

From Tryolabs' previous experience developing a tracking algorithm (Norfair), we knew that we can track objects in real time video streams if the detection model can run at more than 10 FPS.

We can process a 30 FPS video by performing inference once every three frames and then interpolating the bounding box coordinates for skipped frames using the tracker. However, if the model inference runs at lower than 10 FPS, it's harder for the tracker to follow detections across frames, because there is a larger change in position between inferenced frames.

We evaluated the detection accuracy quantitatively on still test images, and qualitatively on video images. This latter was important because we needed to consider the performance on video sequences, leveraging the information of all frames by using the whole pipeline (i.e., after the tracker, score thresholding, and voting system). The qualitative accuracies obtained with MobileNet and YOLO-tiny were similar for the same input resolutions. YOLO-full had considerably better accuracy, even at smaller resolutions, but was too slow in all cases. Only YOLOv4-tiny with a 1024x608 resolution runs fast enough to allow matching detections across frames in real-time, so we selected that model for inference.

Boosting Performance on Jetson Nano: DeepStream

While testing different models, we needed a research-friendly environment for fast prototyping, so we used OpenCV to handle all video processing. All of our codebase and libraries run on Python (including Norfair, Tryolabs' open-source tracking library). However, since we were not running the model inference in parallel with the other needed tasks (e.g., video decoding/encoding, updating the tracker, and drawing), there was a considerable overhead above the bottleneck of 14 FPS for the selected model. This meant that in aggregate, our pipeline would run at only 9 FPS.

Taking this into account, we migrated our whole codebase to run on NVIDIA's DeepStream SDK for intelligent video analytics, which is written in C/C++ but provides Python bindings. This platform uses a different paradigm than our OpenCV approach: all the video processing tasks are implemented using Gstreamer (a software framework for multimedia applications) in which most operations are accelerating using the Jetson Nano's GPU. Python is used to create all the elements in this pipeline and has access to lightweight metadata structures, from which it can read information like detected objects, or provide information in order to create custom drawings into the video pipeline.

Fortunately, an implementation of YOLOv4 is available as a plugin for GStreamer, so integrating our model and the Python tracking and statistics code into this pipeline was reasonably straightforward. Since we only needed access to the detected object's coordinates and classes, all of the costly video processing tasks are handled entirely inside the accelerated pipeline. We only use a high-level language like Python to update the tracker information, collect statistics, and provide the custom drawings for the resulting bounding boxes, as shown in Figure 1.

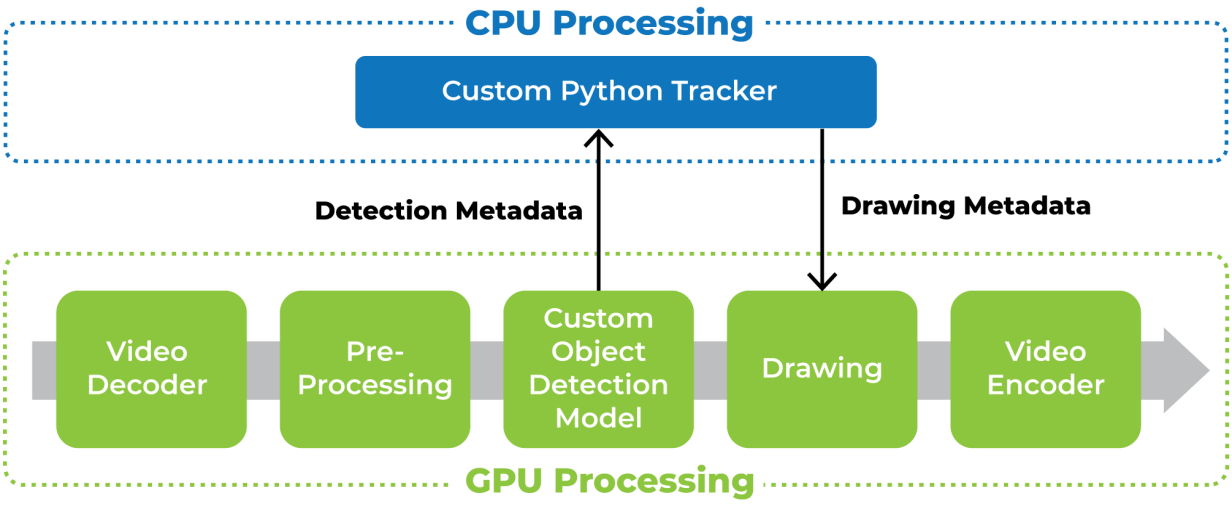

Figure 1: Our custom detector and tracker running as a DeepStream pipeline.

Also, GStreamer uses queues and threads to parallelize the execution on different elements of the pipeline, so we achieved a performance of 14 FPS out of the box, which was the known inference bottleneck. We didn't have to implement any threading on the Python side at this point.

Migrating the inference model to DeepStream and understanding the basics of the GStreamer pipeline took approximately one week. Integrating our Python tracker, voting system, and custom bounding box drawings added another week.

These were the most fundamental building blocks we needed for our application, and migrating them was much easier than expected. The fact that our model was already implemented as a plugin for DeepStream and that we could use the Python bindings to integrate our tracking library were key to this migration.

In the next section, we describe how we overcame the challenges that arose when trying to modify this pipeline dynamically to implement the rest of the application requirements.

Multiprocessing

Although MaskCam's determination of mask/no-mask and the resulting statistics are computed at the edge, our system needs to be able to report this information to a central server; we use the MQTT protocol to accomplish this. Additionally, MaskCam needs to provide a way for camera owners to review videos and detections when required. For the latter requirement, we needed to both activate streaming on demand and save video chunks in the device file system to be downloaded later.

Implementing all of the above using a single GStreamer pipeline turned out to be cumbersome and unstable, mainly because we needed to add and remove elements dynamically without interrupting the video processing.

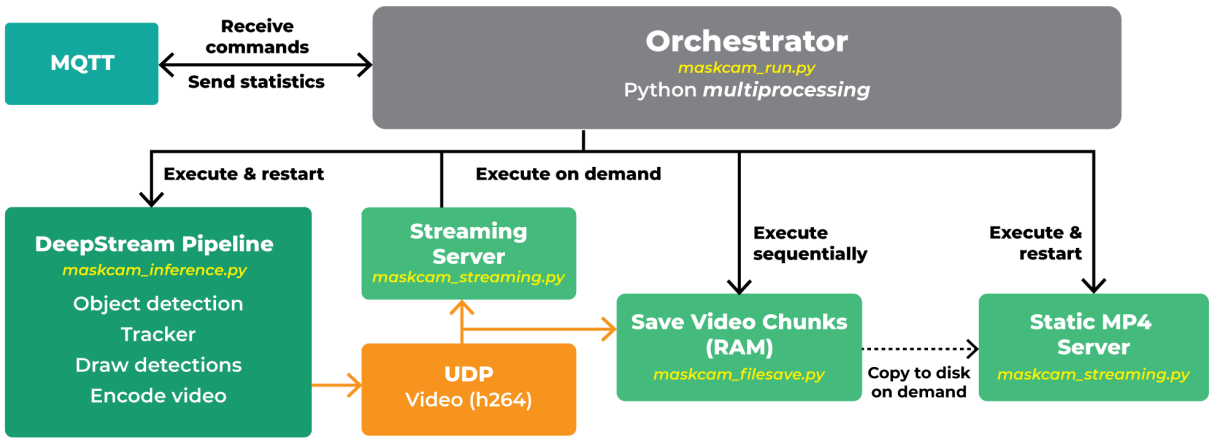

Figure 2: MaskCam software architecture, processes and their relationships.

Instead, we split the code into different processes, as shown in Figure 2. The inference process handles the object detection, tracking, and drawing and outputs video in the form of UDP packets. The streaming and file-saving processes, whenever turned on, read these UDP packets and generate an RTSP streaming server or save them as .mp4 video files. There's also a static file server to allow downloading the saved mp4 videos. These videos are only copied to disk when flagged as important, otherwise they're sequentially saved to RAM and then removed, to avoid wear on the flash storage.

In addition to the above processes we also need to handle MQTT communication, publishing to some topics (stats and device status), and subscribing to others (to receive commands). All of this is done in an orchestrator script (*maskcam_run.py* in Figure 2), which also executes, receives data from, and sends signals to the other processes.

The orchestrator script uses Python's *multiprocessing* module, which enables true parallelization (as opposed to the *threading* module due to the Python GIL) and communication between processes (as opposed to the *subprocess* module).

The orchestrator script supports remote commands to start/stop streaming, restart the inference process, and request to save a video chunk, among others. It's also responsible for executing the file-saving processes sequentially, which saves video files in RAM, and for deleting or moving them to the flash memory upon request. Finally, it sends the statistics from a queue that is filled by the inference process, in order to publish them to the corresponding MQTT topic. (This publishing happens only when an Internet connection is available.)

All of the above-mentioned modules can be run as standalone processes for debugging/development purposes, or entirely managed by the orchestrator script.

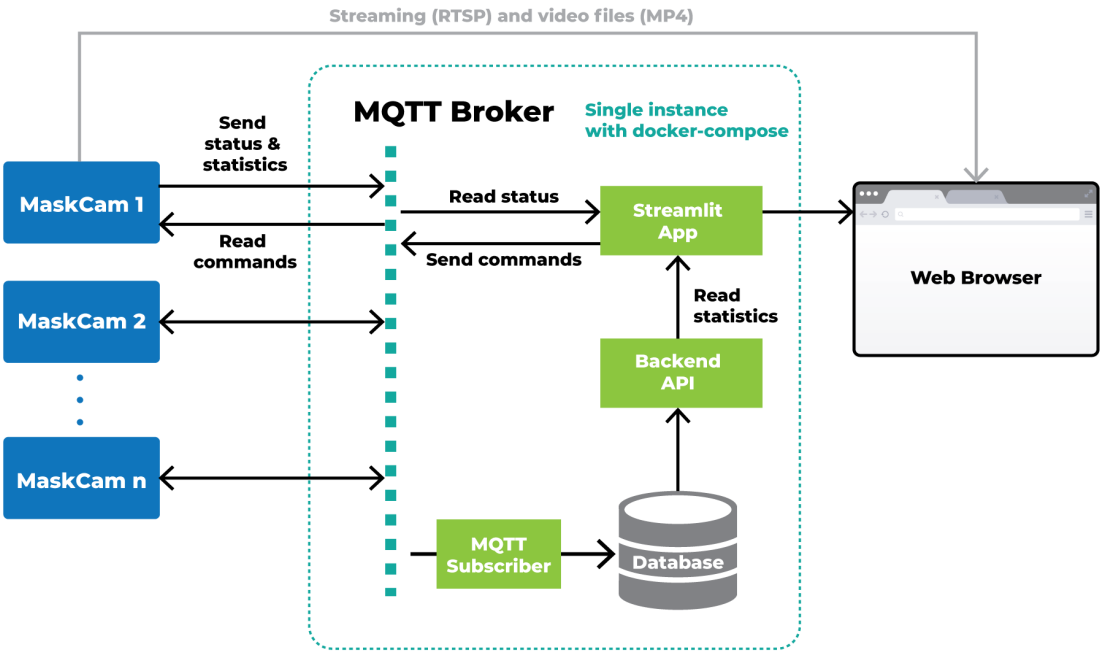

Figure 3: MQTT broker and front-end web server architecture, which runs on a machine separate from the Jetson device.

MQTT Broker and Web Server

To collect statistics and visualize them, we implemented an entirely separate server that runs on a machine other than the Jetson (e.g., on AWS EC2). It receives statistics from the MaskCam device, saves them to a database, and has a web-based GUI frontend to display them. The web frontend is able to send MQTT commands directly to the devices, as well as provide links to visualize streaming and download the video files stored on the device.

The server is implemented using four containers under a docker-compose environment, as shown in Figure 3:

- MQTT broker (eclipse-mosquitto)
  - Receives messages from devices and frontend
- Postgres database (sqlalchemy ORM)
  - Stores all statistics and devices information
- Backend module
  - Runs an MQTT subscriber process which reads all messages from devices and saves them to the database
  - FastAPI server reads from database and provides the information to the frontend

- Frontend app (Streamlit)
    - Displays statistics and links, and sends MQTT commands to devices

The server is implemented entirely in Python (backend API, database model definition, and even the frontend) using the mentioned libraries, and the main functionality required a bit more than two weeks of effort for a senior developer.

We considered using AWS IoT and Greengrass to implement the server, instead of just EC2 instances with containers. In a production environment, it would make sense to consider using AWS IoT Core to implement the MQTT Broker with authentication and other security features, which were out of the scope of our project. We estimated that the front-end, database and API would require similar effort being implemented with AWS tools. Given the roughly equal effort, we elected to remain agnostic of the platform for those services.

Improving the Model and Dataset

Closing the development cycle, we did a final iteration to improve the object detection accuracy by leveraging additional data sets and a face mask detection model developed by NVIDIA. That model uses a combination of four publicly available datasets (Kaggle Medical Masks [6], MAFA [7], FDDB, [8] and WiderFace [9]) to produce one large face mask detection dataset, with approximately 6,000 labels for each object class.

Although this combined dataset was much bigger than our original dataset, it had two major issues: it only contained two object classes (mask, no mask), and since it was a merge of many datasets that were collected for other purposes, it didn't have consistent labelling criteria. For example, some datasets only contained labels of faces wearing masks, and non-masked faces were ignored. One of the datasets only contained one label at the center of the scene, and any other faces were ignored.

We addressed the most obvious issues in this dataset, merged our smaller dataset to provide some samples of *not_visible* and *misplaced* classes, and also added labels for these classes to the new dataset images.

The result was a dataset with 5,223 images, containing 7,008 *mask* labels, 7,866 *no_mask*, 3,678 *not_visible*, and 74 *misplaced* (these latter are harder to find and distinguish, resulting in an unbalanced dataset). Compared to the smaller development dataset, the model trained on this dataset increased its mAP[IoU>0.5] on static images from ~75% to ~90% over all classes, using a random validation set of 10%.

Although the precision for *mask* and *no_mask* objects was improved, the small number of *not_visible* objects causes the system to have a harder time with these objects. As a consequence, the tracker sometimes loses the trajectory for people who are not looking at the camera, but the whole system works better for people whose faces are clearly visible. On the

whole, we feel this model represents both a quantitative and qualitative improvement over our original model.

Next Steps for the Object Detection Model

Since the aim of this project was to provide a smart camera prototype and evaluation of the Jetson Nano (versus creating the most accurate face mask detector possible), there's room for improvement in components like the object detection model and the tracking parameters, whose performance can vary significantly depending on the camera setup (distance, angle and lighting conditions).

For the object detection model, steps that could be taken to improve performance include:

- Curate the dataset more thoroughly, adding all labels for at least the *not_visible* class.
- Find a better balance between model accuracy and inference time, considering detection consistency and the number of frames that the tracker needs to interpolate.
- Provide object visual features to the tracker, to improve matching accuracy.
- Crop frame areas to improve object resolution at the object detector input (evaluate doing it automatically using optical flow)

7. Hardware Design

The next major portion of the MaskCam project was defining the hardware architecture and selecting hardware modules to use for the system.

As mentioned earlier, the Jetson Nano Developer Kit consists of a Jetson Nano processor plus various peripherals and ports (Ethernet, USB ports, GPIO pins, etc.) on a 3" x 4" development board. The Developer Kit is attractively priced ($90, quantity one) but it is neither sold nor warrantied for production use.

For production devices, NVIDIA sells the Jetson Nano module, a small system-on-module (SOM) containing a Jetson Nano processor, 4 Gbytes of RAM, 16 Gbytes of eMMC flash memory, and pins providing various peripheral signals. The Jetson Nano module is $129 in single-unit quantities, with quantity discounts available. The Nano Module must be plugged into a carrier board that provides desired peripherals. For prototyping, carrier boards can be purchased from various NVIDIA ecosystem partners. For higher volume applications we would expect companies to design their own custom carrier board—in essence, a motherboard for their product which the Jetson Nano module plugs into.

The Jetson Nano Product Design Guide from NVIDIA provides detailed descriptions of each hardware interface available, making it easy to understand what devices can be connected. Using this information and the product requirements for MaskCam, we created a hardware block diagram defining the architecture for the system.

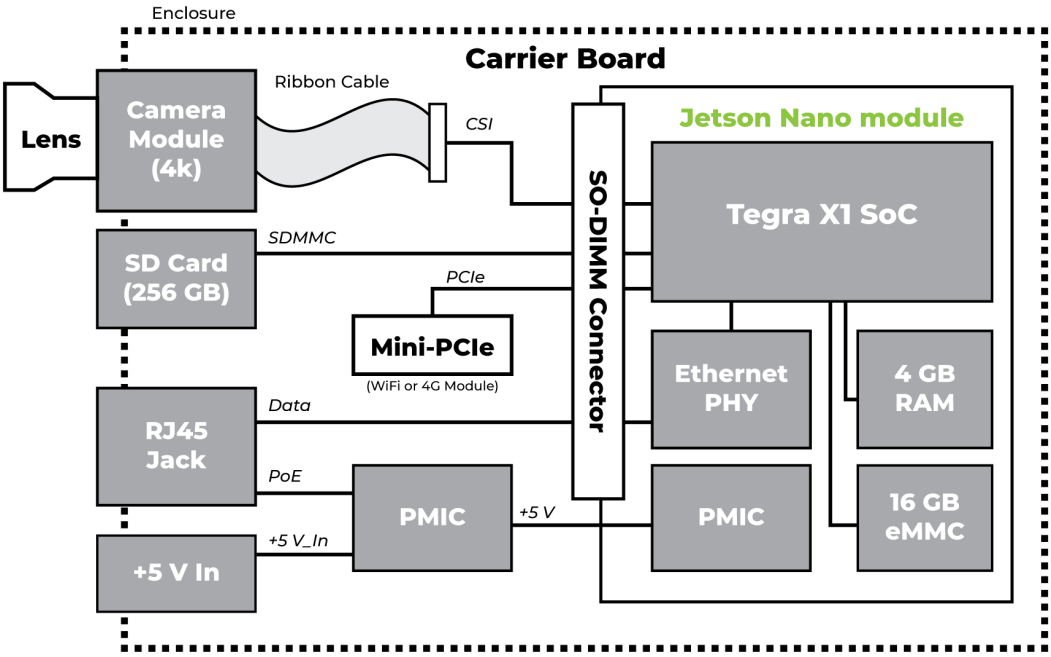

Figure 4: MaskCam hardware block diagram.

As shown in Figure 4, a carrier board hosts the Jetson Nano module and all the peripheral hardware. A 4K camera module connects through a ribbon cable via a MIPI-CSI interface. Internet connectivity may be provided over an Ethernet cable connected to the RJ45 port, or a WiFi or cellular LTE module can be connected to mini-PCIe ports to provide MaskCam with wireless Internet. The system can be powered through a 5-volt adapter or through a Power over Ethernet (PoE) connection.

For each major item in the block diagram we researched options and evaluated them based on their specifications, prices, and support for the Jetson Nano. From this analysis we selected a carrier board, camera module, WiFi module, and LTE module to use for MaskCam.

Carrier Board Selection

The MaskCam prototype uses an off-the-shelf carrier board to simplify development and enable faster time-to-market. For a high-volume version, a full custom hardware solution would need to be designed and manufactured, as discussed in the Productization section below.

Finding a carrier board with the necessary hardware interfaces was made easier by using NVIDIA's Jetson Partner Hardware Products page. It provides a list of carrier board options, and a list of interfaces provided by each option. We selected the Connect Tech Inc. (CTI) Photon [10], which costs $351 per unit, met our design requirements, and had all the hardware interfaces needed for our application. CTI's support team was helpful and responsive, and provided us with a list of WiFi and LTE modules that they had tested with the Photon.

Camera Module Selection

NVIDIA's website provides a list of Jetson Partner Supported Cameras, which shows various camera options that are supported on the Jetson platform. We evaluated several $29 cameras on this list based around the IMX219 image sensor but we were not satisfied with their image quality.

Ultimately we decided to use the Raspberry Pi High Quality (HQ) Camera [11] (which is not on the Jetson Supported Cameras list). It's based around the IMX477 image sensor, which supports 4032x3040 resolution at 30 FPS and 1920x1080 resolution at 60 FPS. Its price point of $75 ($50 camera module and $25 lens) provides a good cost/performance ratio, and it has wide community support due to its popularity. It required a small hardware modification and special drivers from RidgeRun to work with the Jetson Nano, but these were available and well documented through RidgeRun's website.

We found NVIDIA's Jetson Partner Supported Camera list to be a good starting point for researching options, but it could be improved by adding camera prices directly to the list (rather than having to visit each vendor's webpage for pricing) and by providing three to five recommended options at each of three (low, mid, and high) price levels.

Wireless Module Selection

Although MaskCam has wired Ethernet, we realized that not all locations where MaskCam will be installed necessarily have wired Ethernet available. To allow MaskCam to wirelessly connect to the Internet, we decided to include either an optional WiFi module or an optional LTE module. The Photon carrier board provides M.2 B-Key and M.2 E-Key connectors that allow for connecting these modules. The final productized version of MaskCam would include a built-in WiFi chip on the carrier board and an M.2 E-Key slot for the optional LTE module.

For WiFi, our research showed that the Intel 8265 Wifi module [12] isa popular option that is supported by the Jetson Nano. The module has plug-and-play driver support on JetPack OS, and worked immediately with both the Jetson Nano Developer Kit and CTI Photon carrier board. However, it was difficult to get it working with balenaOS, discussed in the Containerization section below.

For LTE, CTI indicated they had tested the Photon with both the Sierra Wireless Airprime EM7455 and Quectel EM06 [13] modules. We selected the Quectel option simply because it was offered at a lower price point ($50 vs. $144). Getting the Quectel module working with the Jetson Nano was difficult and took much longer than expected. While the module itself worked and was immediately supported by the drivers on JetPack OS, making it work with a cellular data plan was challenging. We purchased a 500 MB/month Verizon data plan through Digi-Key's RevX Wireless portal and activated our device's SIM card, but it was not able to establish an Internet connection.

After some troubleshooting, we contacted Quectel for help and they told us the issue was caused by our data plan provider. We then contacted RevX Wireless, who indicated the problem was because our Quectel device wasn't end-certified on Verizon. Quectel acknowledged this and told us to contact Verizon to register our prototype for Verizon's pre-certification process, which required filling out significant paperwork. Ultimately, we switched to an AT&T data plan, and the module was immediately able to connect to the Internet after doing so. The whole process took almost two months of sending back-and-forth emails with various support representatives.

Porting MaskCam from the Jetson Developer Kit to the Photon

The MaskCam code was developed on the Jetson Nano Developer Kit under the JetPack SDK. It was fairly easy to get the MaskCam software running on the Developer Kit using the Developer Kit's built-in Ethernet and a ribbon cable to talk to the Raspberry Pi HQ camera.

With the Photon carrier board and other hardware components selected and in hand, the next step was to bring up the MaskCam application on the new hardware, i.e., on the Connect Tech Photon with a Jetson Nano module. Connect Tech gives instructions for installing the JetPack SDK on the Jetson Nano module with their board support package (BSP) that provides the correct device tree and drivers for the Photon carrier board.

We initially had difficulty getting the Raspberry Pi HQ Camera to work with the Photon. The CTI BSP did not include drivers for the IMX477 sensor used by the camera, and the drivers available through RidgeRun were not compatible with the special Linux kernel on the Photon BSP. Fortunately, mid-way through our project, CTI released an updated BSP that provided support for the IMX477. However, we still needed to manually install a special ISP configuration file to resolve image quality problems with the camera.

Once the issues with the IMX477 driver were resolved, we were able to successfully run the MaskCam program on the Photon.

Thermal Testing

Heat is a big concern for most embedded vision projects, and MaskCam is no exception. In particular, the Jetson Nano processor begins to automatically reduce its clock rate when its on-chip temperature sensor reaches 97° C (206.6° F), a process called thermal throttling. Throttling prevents the system from harming itself but comes at a cost of processing fewer frames per second.

We performed thermal testing with the Photon-based MaskCam application to determine if an active heatsink (i.e., heatsink with a fan) would be needed to prevent throttling. The Photon carrier board was set up in open air on a lab bench while running the MaskCam program while a script continuously polled the Nano's CPU and GPU temperature. This test was repeated for

both a passive and active heatsink. A graph showing the temperature of the CPU and GPU over time for both cases is shown in Figure 5.

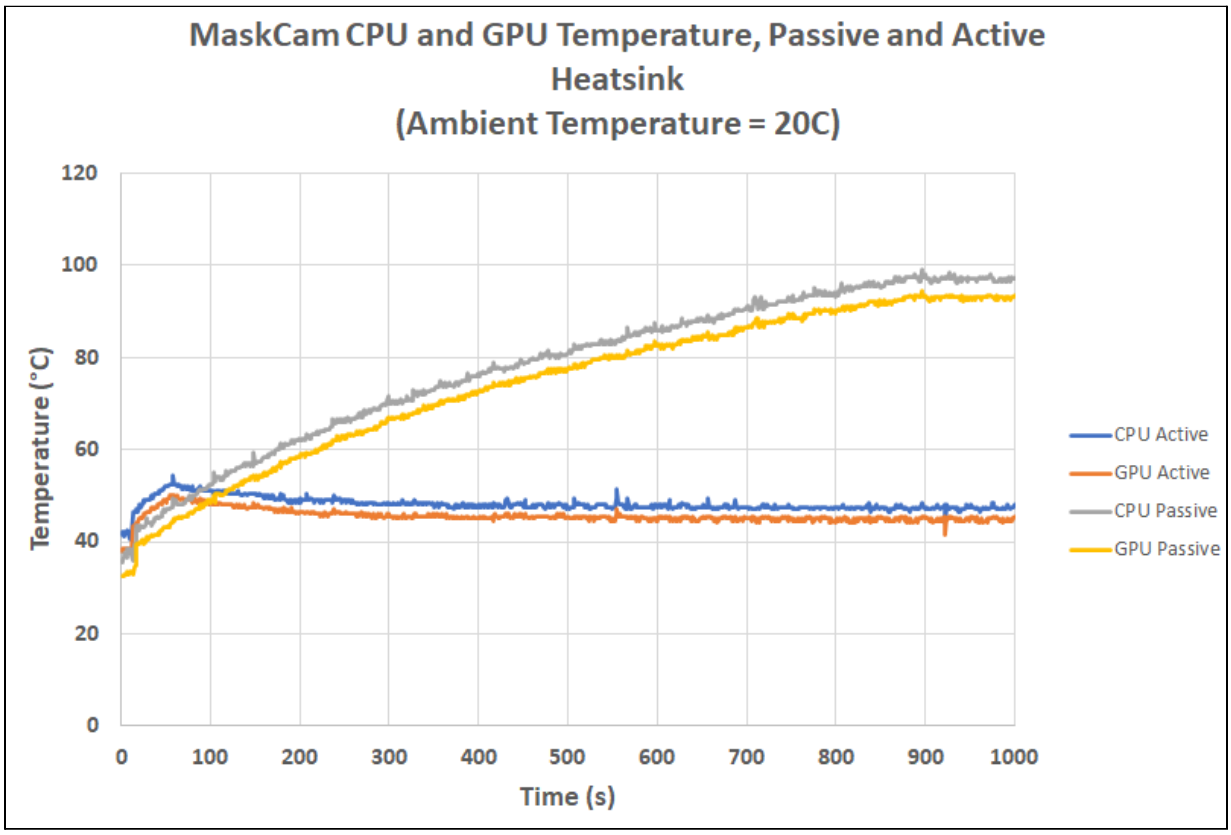

Figure 5: MaskCam GPU and CPU temperature over time with active and passive heatsinks.

The test results showed that the Jetson Nano with a passive heat sink began throttling after about 15 minutes (900 seconds) of operation, demonstrating the need to use a fan in the end product to maintain the Nano at a moderate temperature. As a result, the enclosure design must provide active cooling for the hardware while minimizing the risk of water and dust ingress to the electronics. In the Productization section of this report, we discuss our work with Jabil to create an enclosure concept that meets these requirements.

8. Containerization

For those unfamiliar with the term, a software “container” is a collection of application binaries, libraries, and support files that are supplied as a single file and that run under a host operating system. Containers have several benefits over traditional software installation methods (e.g., “apt install” on Linux):

- They eliminate or greatly reduce version dependencies or conflicts between the application, the host operating system, and other applications by bundling all required files into a single package.

- Since they are self contained, they are easy to distribute and install.
- They provide a level of isolation between containers running on the same host, increasing reliability and security.

Containers have become a best practice for cloud and server applications over the last decade and are just now starting to become popular for embedded applications. Under Linux, Docker is the most popular open source container system and comes pre-installed on the Jetson Nano under NVIDIA's JetPack SDK.

Using Docker containers made development easier for our team in at least two different ways: First, it simplified the installation, build, and test process, since installing a new version of the software with all necessary libraries required only a single command. Second, it made it easier for our group to work on different hardware platforms—specifically, a mix of Jetson Nano Developer Kits and CTI Photon carrier boards with Jetson Nano modules. This was particularly useful because Photon carrier boards were in limited supply during the time we were developing MaskCam, so not all of our team members had access to Photon boards. Using Docker we were able to fashion a single container that worked on both hardware platforms.

The question we grappled with was: would it be feasible to also use containers for production? Happily, the answer was yes. One of NVIDIA's ecosystem partners is balena, which makes a minimal host OS called balenaOS designed for just this purpose. BalenaOS is a very thin Yocto-based Linux distribution built specifically for running containers on embedded devices. Moreover, balenaOS can be used with balenaCloud, a cloud-based fleet-management platform that allows easy configuration, deployment, and updating of balenaOS devices in the field.[1]

We were able to produce a single deployment container image that works not only on the Jetson Nano Developer Kit and Photon hardware, but also on different Linux-based distributions that can run on that hardware: Nvidia's JetPack and balena's balenaOS. This allows users interested in trying out MaskCam to quickly install it on a Jetson Nano Developer Kit with a single command.

Finally, in combination with balenaOS, containerizing MaskCam made for nearly turn-key off-the-shelf updates with all the expected features such as blue/green staged deployments.

CUDA in a Container

The base BalenaOS is not easily modifiable, being designed completely around supporting Docker application containers. While balenaOS provides containers running on the Jetson Nano with full access to the GPU, none of the support libraries such as NVIDIA CUDA Toolkit and DeepStream can be installed on the host OS. In practice this means that our containers are

[1] AWS IoT Greengrass was another possible choice to manage our container deployment, but ultimately we chose Balena; one advantage is that Balena's cloud instrumentation makes it particularly simple to completely upgrade the OS on a host device if necessary.

much larger than average. After CUDA, DeepStream, and Maskcam are installed, our containers are over 6 Gbytes.

While definitely not svelte, proper layering with our code and neural network model being in the last layers of the container make this less of an issue for updates than it might seem. BalenaCloud does a good job of keeping track of layers and only sending necessary differentials to devices.

The one serious concern from a design perspective is that if an early layer change is required (such as a security issue in the base container OS image), it would require the full container to be updated. To update the full container, an entire new container needs to be downloaded and held in storage alongside the existing container. This means that we need slightly more than twice the storage space required by the container itself to accommodate the possibility of doing a full update. An early prototype of our container that was closer to 9 Gbytes sent one of our 16 Gbyte Jetson devices into a fatal out-of-disk-space loop that was not recoverable without reflashing the device.

CTI Photon and balenaOS

Balena supports a wide range of hardware with variations on its base Yocto build. While the Jetson Nano Developer Kit hardware is officially supported by balena, balenaOS for the CTI Photon carrier board is a community build maintained by Drebble, an edge computing solutions company based in the Netherlands.

We built a balenaOS image for the CTI Photon Carrier board using the balenaCloud dashboard and installed it on our Jetson Nano module. Using balenaOS lets us connect to the device and interact with it remotely through a web-based balena dashboard. The balena dashboard provides information about the device and allows us to quickly download the MaskCam container from balenaCloud onto the device. It provides device and container configuration options, with one interesting feature being the ability to point to different device tree files on a per-device basis. We used this setting to enable the correct DTB file for running our camera. The screenshot in Figure 6 shows how the device appears on the dashboard.

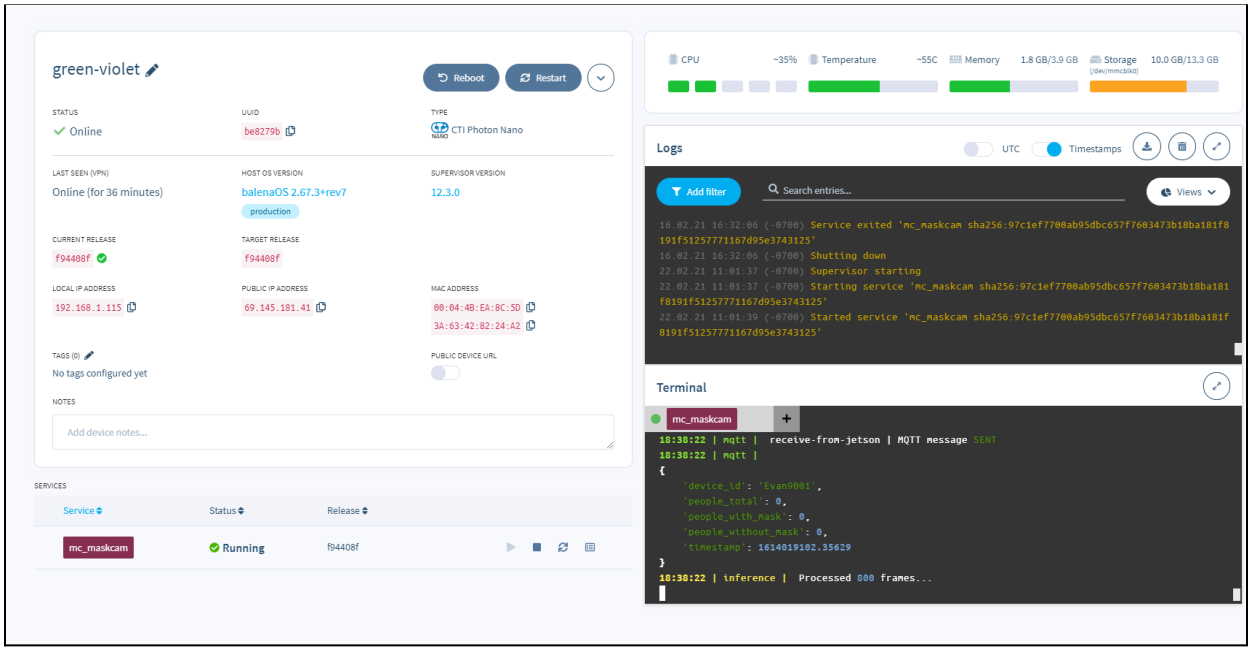

Figure 6: Device information on balenaCloud dashboard.

Getting the Raspberry Pi HQ Camera and Intel 8265 WiFi modules up and running on the CTI Photon carrier board under balenaOS turned out to be somewhat challenging.

The CTI Photon build from Drebble was lagging behind the CTI kernel driver package and BSP, which meant that it didn't have the driver patches needed to access the Raspberry Pi HQ Camera. After conversations with Drebble and balena, the problem was rectified rapidly, and we installed the right DTB and drivers for the camera. We were impressed with the level of support we received from the engineers at Drebble for their open source build.

The Intel 8265 WiFi module did not immediately work with balenaOS like it did on JetPack OS. BalenaOS had specific drivers for the 8265 module that were also integrated into the CTI Photon build. However, a bug with the Active State Power Management (ASPM) feature of the PCIe driver prevented the module from initializing correctly. We had to disable ASPM in the boot configuration to get it to work. Again, the support from balena and Drebble was very helpful here.

9. Productization

With a functioning prototype in hand, we turned to Jabil Optics to get an estimate of MaskCam manufacturing costs for volume production. Jabil is one of the world's largest electronics manufacturers, providing manufacturing services for Apple, Cisco, Dell, HP, and many others; Jabil Optics is a specialized group within Jabil that provides design and assembly services for optical electronics, including smart camera designs.

We worked with Jabil Optics engineers to estimate the overall hardware BOM cost, explore options for custom camera modules, get a preliminary estimate for the cost of a custom enclosure, and to understand manufacturing value added costs. Our target quantities were 10K and 100K units/year.

We provided Jabil with the hardware specifications for MaskCam and a list of the off-the-shelf hardware modules used in the prototype. NVIDIA also provides an open source schematic of the Jetson Nano Developer Kit, which our team found to be helpful for accelerating the hardware design timeline. Jabil utilized the open source NVIDIA schematics (and the schematics for another open source board, the Antmicro Jetson Nano Baseboard) as a starting point to determine the production BOM.

We kept the requirement for a mini-PCIe connector for an optional LTE module, but we moved the optional mini-PCIe WiFi interface onto the custom carrier board using a low-cost 802.11b/g/n WiFi chip.

Jabil Optics removed unnecessary interfaces (e.g., the HDMI interface) from the schematics and then estimated the volume cost for the remaining components.

Jabil Optics researched various image sensors and lenses that could be used for a custom camera module at a lower cost point than the Raspberry Pi HQ Camera. They found image sensors that met our application requirements while having a better cost to performance ratio than the IMX477 sensor. They also considered off-the-shelf and custom lens designs for these sensors that provide a wide field of view and met the other sensor requirements (image format, resolution, etc.). Ultimately, their proposed custom camera module cost $34 at 100K volume. This camera module would need to be designed before entering manufacturing production of MaskCam.

We asked Jabil Optics to investigate the possibility of using an HD-resolution sensor instead of a 4K sensor. To our surprise, they determined that the estimated difference in cost, including both sensor and lens, would only be about $4.

A critical aspect of MaskCam that needs to be completed before entering manufacturing production is the enclosure design. Jabil has design services for developing custom enclosures and they worked to estimate its cost. The enclosure has several unique requirements:

- Industrial design of case must allow for mounting the MaskCam unit in various locations.
- The enclosure must provide some level of weather protection for outdoor installations of MaskCam.
- The enclosure must provide active cooling and air flow for the Jetson Nano to prevent it from throttling.

To meet these requirements, Jabil proposed a finned metallic enclosure that acts as a heatsink for the Jetson Nano. The enclosure has two compartments. One interior compartment contains

the Jetson Nano and other sensitive electronics and is IP67 rated to prevent water and dust ingress. The Jetson Nano is thermally linked to the enclosure through thermal grease to promote heat conduction from the Nano to the enclosure. A second compartment contains a fan and air channel for actively cooling the fins of the enclosure. The estimated cost for this enclosure is $32 at a 100K volume.

Finally, the MaskCam product has to be manufactured, assembled, and packaged in volume. Jabil's manufacturing value added (MVA) costs can range from 25% - 50% of the overall BOM cost, depending on:

- Production volumes (which impact efficiencies and amortization)
- Design customizations required
- Custom manufacturing equipment required
- Location of manufacturing - total system cost (supply chain, equipment availability, tariffs, etc.)

Jabil estimated an initial MVA of 40% for producing MaskCam in quantities of 100K/year. Table 4 shows a breakdown of MaskCam's estimated production cost at 10K and 100K volumes. The table also includes pricing for optional modules, such as a 256 Gbyte SD card or an LTE module.

Item	Description	Cost (USD, 10K/year)	Cost (USD, 100K/year)
Camera Module	4K (e.g., Sony IMX215)	$42.36	$33.84
Carrier Board	See Figure 4	$54.82	$53.13
Jetson Nano Module		$99.00	$89.00
Enclosure, Fan, WiFi ant.		$41.20	$35.29
Total BOM		$237.38	$211.26
MVA	50% @ 10K, 40% @ 100K	$118.69	84.51
Total Factory Out Price		**$356.07**	**$295.77**
Optional:			
SD Card, 256 Gby	E.g., SanDisk Class 10	$30.00	$26.00
LTE M.2 Card	E.g., Huawei ME0909S	$50.00	$45.00
Ethernet Cable	Cat 6	$1.00	$1.00
IP67 Connector Cover		$2.50	$2.00
Power Supply	5 V, 4 A	$5.00	$4.00

Table 4: Jabil Optics' cost estimates for MaskCam and its optional components at quantities of 10K and 100K per year.

Jabil emphasizes that the costs shown in Table 4 are estimates, and will be impacted by such factors as:

- Industrial design of case
    - Materials
    - Mounting requirements
- Lens design
    - “Off-the-shelf” vs. “custom”
    - Sacrifice optical performance for cost
        - Field of view, modulation transfer function, chief ray angle
    - Lens environment (e.g., do we need to prevent fogging of lens?)
- Camera module design
- Board design
- Thermal design
    - Link heat sink to case through thermal grease/pad to promote conduction
    - Fan design and fan port design (if required)
- Packaging
- Manufacturing location
- Legal issues
    - Liability - who is responsible? (Jabil, customer, etc.)
    - Representations and warranties

10. Steps to Production

After completing the efforts to containerize the MaskCam application and deploy it on the CTI Photon carrier board, we have a fully functional prototype of the mask detection camera. BalenaCloud allows us to quickly load the container on the Photon-based Jetson Nano, configure it, run the MaskCam application, and deploy updates. The camera is able to monitor crowds and report statistics on mask detections to a cloud-based server using MQTT. A browser-based front end allows users to view statistics and remotely download video clips. A live stream of the camera feed, with detections drawn on each frame, can be viewed over a local network (Figure 7).

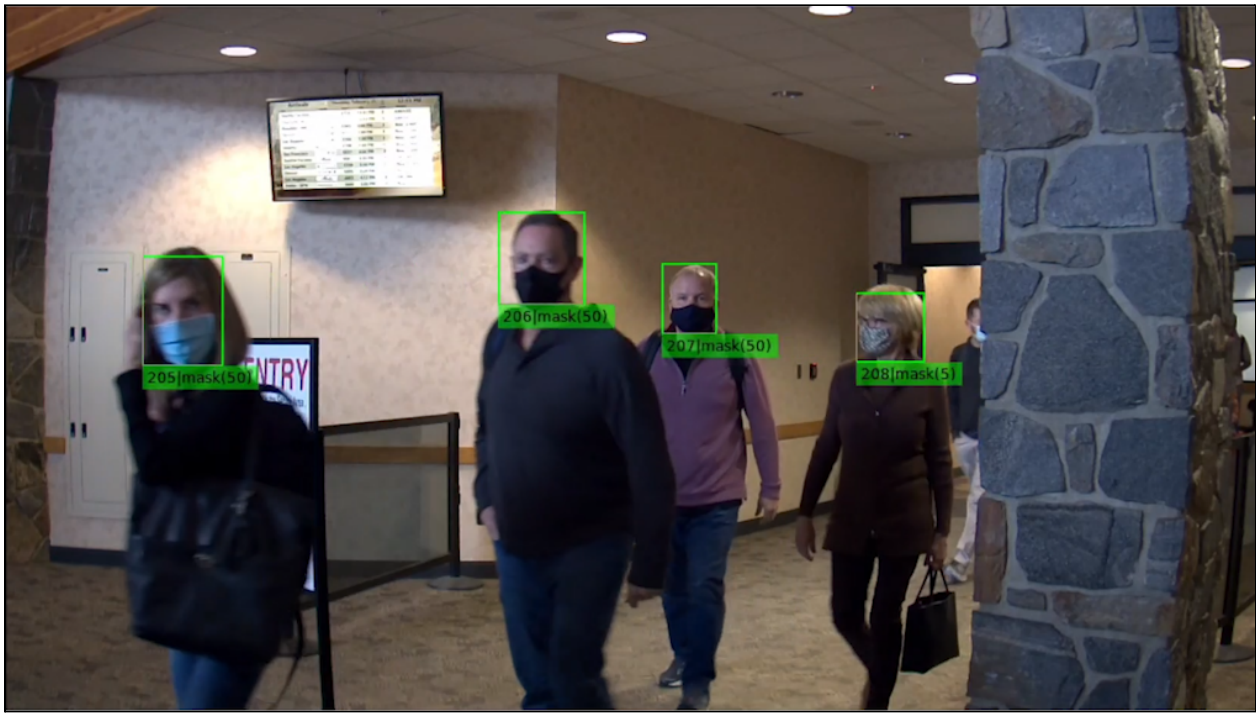

Figure 7: MaskCam in action at an airport.

There are a number of tasks to complete before MaskCam could enter production:

- **Enclosure.** An enclosure would need to be created to encapsulate the hardware, protect the electronics from water and dust, and allow for adequate cooling. To start with, this could be a basic enclosure for the rapid time-to-market iteration of our product (the one that uses off-the-shelf hardware modules, like the Photon carrier board), and then could be followed by a more elaborate enclosure for our high volume product. Further thermal testing would be needed to characterize the performance of any enclosure or thermal solution.

- **Carrier board design.** We have sketched out the components that we believe we would need on our custom carrier board, and Jabil Optics has produced manufacturing cost estimates based on this sketch, but we have not actually designed or manufactured a custom carrier board.

- **Set up scripts.** We have not addressed the set-up scripts that would allow a new MaskCam owner to easily get a MaskCam device set up on their network. For WiFi these would typically be set-up scripts that would allow the MaskCam to start out of the box as a WiFi access point and allow a user to configure network credentials; for wired Ethernet this might be a smartphone app that displays a configuration QR code that MaskCam could read through its camera. For the former at least, balena provides a "WiFi Connect" github repo.

- **Quality assurance and compliance testing.** MaskCam needs to be tested to characterize its accuracy in a variety of visual conditions. We need to confirm that the camera is able to operate, stay connected to the remote web server, and report data for weeks or months without human intervention. Environmental testing is needed to verify MaskCam is able to operate in outdoor conditions, as well as compliance testing to demonstrate MaskCam meets regulatory requirements set by organizations like the FCC.

Additionally, with any neural-network based product, there is room for improvement in the accuracy and implementation of the deep-learning algorithm. As mentioned in the Software Development section, our mask detection model could be improved with a larger dataset, particularly with images of people whose face is not visible. Additionally, one could experiment with different models (such as NVIDIA's PeopleNet, or YoloV5) to improve the accuracy, at a tradeoff of lower frame rate. Finally, it would also be possible to use motion detection to detect and crop regions of interest of each frame and pass these smaller size images to the model to improve model accuracy for people at a distance.

11. Conclusions

Overall, our team was impressed with the Jetson Nano and its ecosystem. The quality documentation and examples for the NVIDIA SDKs, the breadth of the hardware partners and modules available to be used with the Jetson Nano, and the containerization tools from balena all facilitated rapid development of our smart mask detection camera. The fact that we were able to conceptualize, design, and create a production-ready prototype of MaskCam in a short time with a small team speaks to this.

That said, we encountered several challenges in both the hardware and software development efforts. We ran into two general categories of problems:

- **Discoverability.** NVIDIA and its ecosystem move quickly, with frequent updates to hardware, software, and drivers, and much of the support is provided via the NVIDIA Developer Forum. In general, this is a good thing! But it means that sometimes it is hard to discover solutions to problems. For example, does a given peripheral work on the Jetson Nano? Doing a quick Google search may lead you to an outdated forum posting from six months ago saying that it doesn't—but that hasn't been updated to reflect that, as of a month ago, it now does.

- **Switching platforms.** Generally, we found the process of moving from one platform to another to be difficult and time consuming. While developing our application on the Jetson Nano Developer Kit was easy enough, in porting it to new hardware or software platforms we often ran into problems. On the hardware side, we had difficulty porting the MaskCam application from the Developer Kit to the Photon carrier board. On the software side, we encountered difficulty when porting from JetPack to balenaOS and

Docker containers. NVIDIA could improve the overall ecosystem by making sure their hardware and software partners coordinate to support hardware and software platform interchangeability, but this is a tall order given the size of the ecosystem and the pace at which things move.

Here are some more specific examples of the issues we worked through while developing MaskCam:

- The documentation for using DeepStream with custom models and Gstreamer Python bindings was lacking, which created a steep learning curve on that aspect of the port to DeepStream.
- We struggled with getting hardware module drivers to work on the Photon carrier board, both with JetPack OS and balenaOS. These included the Raspberry Pi HQ camera, the Intel 8265 Wifi module, and the Quectel LTE module. While there's plenty of documentation and instructions on using the Jetson Nano Developer Kit, the level of support drops off very quickly when moving to the Jetson Nano module on an independent carrier board.
- The 16 Gbyte storage limit of the Nano's eMMC chip was difficult to work with. We spent significant time reducing the size of our container and application to get it under 8 Gbytes (in order to provide space for in-field updates).
- Containerizing CUDA and DeepStream under balenaOS was challenging because the usual method of installing packages (e.g., apt install) did not provide a working installation. There was a lack of package dependency information, and the error messages from running sample code gave no information on which packages were missing.

Fortunately, by searching the NVIDIA Support Forums and receiving help from NVIDIA's partners, we were able to overcome these problems.

In conclusion, we found the Jetson Nano to be a useful enabling technology for edge applications requiring significant processing power. The NVIDIA hardware and software ecosystem accelerates the timeline for developing intelligent devices. There are several pain points with using the Jetson Nano, particularly with porting applications to new platforms, but these are alleviated by NVIDIA's support community. While the Jetson Nano's price and power consumption will not fit all applications, for those it does, the Nano merits consideration by product teams looking to rapidly develop edge AI processing systems.

We invite readers to try out the MaskCam code, available at https://github.com/bdtinc/maskcam; you can run it in a container on a Jetson Nano Developer Kit with a USB webcam with just two commands, as described in the repo's README. Questions may be directed to maskcam@bdti.com.

12. References

Jetson documentation:
[1] *Jetson Module Technical Specifications:*
https://developer.nvidia.com/embedded/jetson-modules
[2] *Jetson Nano Developer Kit:*
https://developer.nvidia.com/embedded/jetson-nano-developer-kit
[3] *JetPack SDK:* https://developer.nvidia.com/embedded/jetpack

Mask detection algorithm documentation:
[4] *Face mask detection in street camera video streams using AI: behind the curtain:*
https://tryolabs.com/blog/2020/07/09/face-mask-detection-in-street-camera-video-streams-using-ai-behind-the-curtain/

Yolo V4 documentation:
[5] *Real-time object detection method for embedded devices:*
https://arxiv.org/ftp/arxiv/papers/2011/2011.04244.pdf

Face mask datasets:
[6] *Kaggle Medical Masks Dataset:*
https://www.kaggle.com/shreyashwaghe/medical-mask-dataset
[7] *MAsked FAces (MAFA) Dataset:*
http://221.228.208.41/gl/dataset/0b33a2ece1f549b18c7ff725fb50c561
[8] *Face Detection Dataset and Benchmark (FDDB):* http://vis-www.cs.umass.edu/fddb/
[9] *WIDER FACE Dataset:* http://shuoyang1213.me/WIDERFACE/

Jetson ecosystem documentation:
[10] *Connect Tech Photon carrier board:*
https://connecttech.com/ftp/pdf/CTIM_NGX002_Manual.pdf
[11] *Raspberry Pi High Quality Camera:*
https://www.raspberrypi.org/products/raspberry-pi-high-quality-camera/
[12] *Intel Dual Band Wireless-AC 8265 module:*
https://ark.intel.com/content/www/us/en/ark/products/94150/intel-dual-band-wireless-ac-8265.html
[13] *Quectel EM06 cellular module:* https://www.quectel.com/product/EM06.htm

