CROESUS: MULTI-STAGE PROCESSING AND TRANSACTIONS FOR VIDEO-ANALYTICS IN EDGE-CLOUD SYSTEMS

Samaa Gazzaz[1], Vishal Chakraborty[2], and Faisal Nawab[2]

[1]University of California, Santa Cruz
[2]University of California, Irvine

ABSTRACT

Emerging edge applications require both a fast response latency and complex processing. This is infeasible without expensive hardware that can process complex operations—such as object detection—within a short time. Many approach this problem by addressing the complexity of the models—via model compression, pruning and quantization—or compressing the input. In this paper, we propose a different perspective when addressing the performance challenges. Croesus is a multi-stage approach to edge-cloud systems that provides the ability to find the balance between accuracy and performance. Croesus consists of two stages (that can be generalized to multiple stages): an initial and a final stage. The initial stage performs the computation in real-time using approximate/best-effort computation at the edge. The final stage performs the full computation at the cloud, and uses the results to correct any errors made at the initial stage. In this paper, we demonstrate the implications of such an approach on a video analytics use-case and show how multi-stage processing yields a better balance between accuracy and performance. Moreover, we study the safety of multi-stage transactions via two proposals: multi-stage serializability (MS-SR) and multi-stage invariant confluence with Apologies (MS-IA).

***K*eywords** multi-stage transaction, object detection, performance, accuracy

1 Introduction

Modern object detection models are based on complex Convolutional Neural Networks (CNN) that require GPU clusters costing tens of thousands of dollars to perform object detection in real-time [1–4]. This is infeasible for edge applications that require real-time processing but cannot afford to place expensive hardware at the edge. Furthermore, many of these applications require response in the scale of milliseconds (such as V/AR [5] and smart city Vehicle-to-Everything [6]). This prohibits the use of faraway cloud resources.

There is a large body of research in the machine learning community that aims at addressing the trade-off between accuracy and performance in deep learning (DL) models by utilizing compression, pruning and quantization techniques [2–4, 7–14]. In these approaches, we notice a trade-off between accuracy and performance. The accuracy of a compressed model is typically lower compared to the full model while performance is improved dramatically. For example, in [2], the compressed model improves latency from 23.1 ms to 2.9 ms, while lowering the accuracy from 74.1% to 50.2%. Other papers in the field of image compression aid in reducing the amount of time needed to process data [15–17]. other researchers opt to specializing DL models for certain use cases to improve performance [18–21].

An important aspect that is overlooked in many video analytics solutions is that they are not integrated with the system's data processing and management. Video analytics generates insights from videos that would typically be used in a data management application. For example, detecting objects in V/AR might feed into a mobile game, immersive social network, or other application. We propose Croesus, a multi-stage edge-cloud video processing framework that aims to manage the performance-accuracy trade-off in DL models. The framework consists of an edge-cloud video analytics component and a transaction processing component. Each component may exist in isolation of the other and benefit other use cases, however, they are co-designed to achieve the goals of data management for video analytics applications.

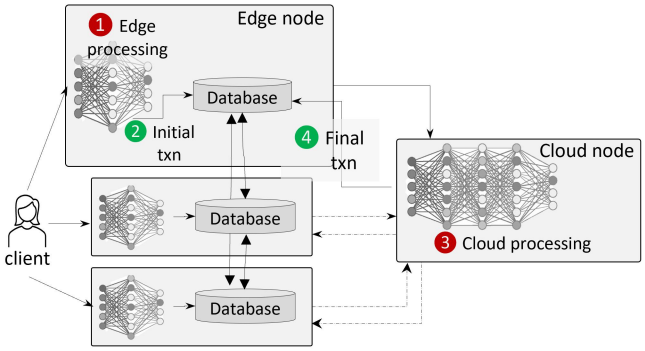


Figure 1: Croesus' execution pattern

This proposal separates computation into two stages: an initial stage that depends on best-effort computations at the edge (using a fast but less accurate DL model), and a final stage at the cloud to correct any errors incurred in the initial stage (using the accurate but slower DL model.) For example, for object detection in applications such as V/AR, instead of depending solely on the full CNN model, a more compact model is used at the edge to respond immediately to users. If needed, some frames are sent to the full CNN model on the cloud to detect any errors on the immediate responses sent by the initial stage. If an error is detected, then a correction process is performed in the final stage. The mechanism to correct errors is an application-specific task and our method allows flexibility in how errors are corrected. The advantage of this model is that users have the illusion of both a fast and accurate object detection. The downside is the possibility of short-term errors. This pattern of the multi-stage model is useful for applications that require fast response but where the full model cannot be used within the desired time restrictions.

We formalize and analyze the transactions (a transaction is a group of database read/write operations that represents a task or a program) in Croesus using a formal *multi-stage transaction* model. Our model divides transactions into two sections: an initial and a final sections (we also show how this model can be extended to multiple sections). The initial section is responsible for updating the system using the results of the initial object detection stage, and the final transaction is responsible for finalizing/correcting state using the results of the final (object detection) stage. The multi-stage transaction model can be generalised to have more than two stages. However, our analysis with the general design turned out to add additional overhead without providing a significant benefit for edge-cloud video analytics. The reason is that the asymmetry in edge-cloud systems is two-fold: in the edge (low-capability, real-time requirement) and in the cloud (high-capability, less stringent latency requirement).

The multi-stage transaction model leads to challenges when reasoning about the correctness guarantees that should be provided to users. This is because the multi-stage transaction model breaks a fundamental assumption in existing transaction models, which is the assumption that a transaction is a single program or block of code. Therefore, there are challenges on coming up with an abstraction of initial and final sections and how they interact. Also, there is a need to specify what makes an execution of initial and final sections correct in the presence of concurrent transactions. We cannot reuse existing correctness criteria—such as serializability [22]—as they would not apply to the multi-stage transaction model.

For those reasons, we propose a multi-stage transaction processing protocol and study the safety-performance trade-offs in multi-stage transactions. We investigate two safety guarantees: *(1) Multi-stage Serializability (MS-SR)*, which mimics the safety principles of serializability [22] by requiring that each transaction would be isolated from all other transactions. *(2) Multi-stage Invariant Confluence with Apologies (MS-IA)*, which adapts invariant confluence [23] and apologies [24] to the multi-stage transaction model and enjoys better performance characteristics and flexibility compared to MS-SR. The multi-stage transaction pattern of Croesus invites a natural method of adapting invariant confluence and apologies. In particular, the final section is—by design—intended to fix any errors caused by the initial stage. This can be viewed as the final stage "correcting any invariant violations" and issuing "apologies" for any erroneous work generated by the initial section.

In the rest of this paper, we present background in Section 2, followed by the design of Croesus (Section 3) and multi-stage transactions (Section 4). Experiments and related work are presented in Sections 5 and 6, respectively. The paper concludes in Section 7.

2 Background

In this section, we present background on the multi-stage system model and object detection.

2.1 System and Programming Model

Edge-Cloud Model. Our proposed system model consists of edge nodes and a cloud node (see Figure 1). Each edge node maintains the state of a partition (database transactions are performed on the partition copy at the edge.) For ease of exposition, we focus on a single edge node and partition in this paper. In edge applications, interactions between users tend to have spatial locality and are therefore typically homed in the same edge node and partition.

Application Model. The applications we consider are video-driven—meaning that the input and triggers to operations on data are done via a video interface. For example, a gesture or object detected on a V/AR headset triggers a database transaction. This translates to the following processing steps for each frame f: (1) the frame f is processed using the small model on the edge node, M_e, to generate labels (labels are the detected objects and/or actions). We call these the edge labels and are denoted by a set L_e. (2) the edge labels L_e are used to trigger transactions that take the labels as input. These transactions are denoted by the set T_f. The initial sections of each of these transactions in T_f are processed to return an immediate response to users and potentially write to the database on the edge node. (3) concurrently, the frame f is also processed in the original, more accurate object detection model on the cloud, denoted by M_c. Once the cloud model generates the labels, denoted by L_c, they are sent to the edge node. (4) when the labels L_c from the cloud are received, they are used to trigger two types of events. The first is to trigger the final sections of the transactions T_f that started for frame f. The input to these sections is the correct label(s) of the object(s) that triggered the transaction. The second is to trigger new transactions that should have been triggered by the frame but their labels where missing in L_e. We focus on the first pattern as the second pattern can be viewed as a subset of the first.

Example Application. Consider a smart campus Augmented Reality (AR) application with two basic functionalities: (1) Task 1: continuously, an object detection CNN model detects buildings in the campus. If a building is detected, the database is queried and information about the building—such as available study rooms—is augmented onto the headset view. (2) Task 2: if the user clicks on an auxiliary device, a study room is reserved in the currently detected building.

Execution Pattern. The execution pattern of this application is the following (shown in Figure 1): The headset captures images continuously and sends them to the nearby edge node. The edge node performs the initial stage of computation by running the captured frame, f on the small (fast but inaccurate) DL model, M_e (step 1). The labels extracted from the model, L_e, are used to trigger the initial section of transaction T_f (step 2). For example, if the engineering building is detected, then the transaction's initial section reads information about the building. The outcome of this transaction is sent back to the headset to be rendered and augmented onto the display. During this time, the frame is forwarded to the cloud node which runs the full (slow but accurate) CNN model, M_c (step 3). The labels, L_c extracted from the model are sent back to the edge node. Once the edge node receives the correct labels, it performs the final stage of the transactions in T_f (step 4). The final stage takes as input both the original detected labels in the initial stage as well as the new, correct, labels.

Programming Interface. The programming model exposes an interface to write both the initial and final sections of the transaction. In our application for example, there are two transactions, one for each task. For task 1 (display information about detected buildings), the initial section is triggered for each frame with a label in the class "building" and it takes as input the detected labels, L_e. For each detected label, the initial section reads the information about that key from the database and returns it to the headset to be rendered. The final section is triggered after the correct labels, L_c, are sent from the cloud node. It checks if the labels are the same; if they are, the transaction terminates (note that the decision to terminate is specific to this example transaction, but other application might use the final section to perform some final actions even if the labels were correctly detected in the initial stage.) If they are not, then the transaction reads the labels of the correct detected building and sends them to the headset to render the correct information and an apology. [1]

For task 2 (reserve a study room), the initial section is triggered when the auxiliary device is clicked by the user. The initial section takes as input the most recent detected labels and their coordinates. If there are more than one label, the initial section picks the label that is closest to the center of the frame. Then, the initial section reserves a study room if one exists. The final section—triggered after receiving the correct labels—checks if the center-most label matches the building where the study room was reserved. If so, the transaction terminates. Otherwise, the original reservation is removed from the database and—if available—a new reservation with the right building is made. The results are sent back to the AR headset to be rendered with an apology.

[1] In a real application, the corrected information would also influence the small model—via retraining and heuristics such as smoothing—so that the error would not be incurred in the following frames.

2.2 Accuracy-Performance Trade-off in Object Detection

Convolutional Neural Networks (CNNs). A CNN is designed and trained to detect labels of objects in an input frame. Different CNN models have different structures and variations, and we refer the interested reader to these surveys [25, 26]. Our work applies to a wide-range of CNN models as we use them as a black box.

Accuracy-Performance Trade-off. The complex processing of CNNs result in higher inference time. It is estimated that running a state-of-the-art CNN model in real-time requires a cluster of GPUs that costs tens of thousands of dollars [1]. This means that running a CNN model on commodity hardware—such as what is used in edge devices—would lead to prohibitively high latency. This led to exploring the accuracy-performance trade-off in CNN models. Specifically, there has been efforts to produce smaller CNN models that would run faster on commodity hardware [20, 27–31]. The downside of these solutions is that they are less accurate than full CNN models. In this work, we aim to utilize both small and full CNN models by using small models for fast inference and original models to correct any errors.

Derivative Models. The interest in the accuracy-performance trade-off in CNNs led to efforts that enable deriving smaller—faster—models using existing original CNN models. One approach is to use a smaller model that handles the same scope of labels of the original model but with less accuracy [27]. Another approach is to create smaller—specialized—models that narrow the scope of labels to enable faster inference while retaining accuracy for the select labels [1]. In our work, we consider both variations. For smaller, less accurate models, the Croesus pipeline helps correct errors due to inaccuracy and for specialized models, the Croesus pipeline helps correct errors due to the narrower scope of labels.

3 Croesus Design

In this section, we present the design of Croesus and an optimization that controls the accuracy-performance trade-off.

3.1 Overview

System Model. The system model of Croesus (Section 2) consists of an edge node and a cloud node. The edge node hosts a small CNN model denoted by M_e that is used to perform initial processing. The edge node also hosts the main copy of it's partition's data. The edge node processes both the initial section and the final section. The initial section of a transaction is triggered by the labels of the model on the edge, M_e, and the final section is triggered by the labels of the model on the cloud, M_c. The execution pattern of requests is shown in Figure 1 and described in Section 2.1.

Workflow. The workflow of requests in Croesus is the following: a frame f is sent from the client to the edge node. The edge node processes f using the edge model, M_e. The labels from M_e, L_e, are used to trigger corresponding transactions, T_f (the programmer defines what transactions should be triggered for each class of labels.) The initial sections of transactions in T_f are processed on the edge node. At this time, the response from the initial sections are sent to the client. This marks the initial commit stage. In the meantime, the frame f is sent to the cloud node. Once the cloud node receives it, the cloud model, M_c, is used to process f. The corresponding labels, L_c, are then sent to the edge node. When the edge node receives the cloud labels L_c, the final sections of transactions in T_f are triggered. The responses and apologies from these final sections are sent to the client. This marks the final commit stage.

Bandwidth Thresholding. The pattern of edge-cloud stages introduces a bandwidth overhead due to the need to send all frames from the edge to the cloud. This can be problematic due to the high overhead on the edge device and the monetary cost of communicating data to the cloud. (e.g., some public cloud providers charge a cost for communicated data between the data center and the Internet). To this end, we tackle the problem of limiting edge-to-cloud communication. We use the confidence of the labels that are generated by the edge model, M_e, to decide whether we need to send the frame to the cloud or not. Specifically, if the edge model's confidence is high enough, this is an indication that the detected labels are more reliable than other detections that have less corresponding confidence. Later in this section, we develop a bandwidth thresholding mechanism to investigate sending frames to the cloud selectively using the edge model's confidence.

3.2 Initial-Final Section Interaction

A unique property of multi-stage processing is that there are two stages of processing where the first stage is fast and less accurate and the second is slow and accurate. This property leads to the need to understand how they interact and what guarantees should be associated with each stage. In the rest of this section, we provide such properties that are useful to programmers in the multi-stage model. In the initial stage, the initial section of a transaction, s_i, uses the input from the edge model, M_e, to generate a response to the user. This response represents an *initial-stage commit*. The

initial-stage commit—when received by a client—represents the following: (1) the response is a preliminary and/or best-effort result of the transaction. (2) any errors in this initial processing will be corrected by the logic specified by the programmer in the corresponding final section. This second property is critical because it leads to having to enforce a guarantee that if the initial section of a transaction returns a response to the client (an initial-stage commit), then the underlying system must guarantee that the corresponding final section would commit as well. This is trivial for a transaction running by itself, however, when transactions are running concurrently, this leads to complications. (In Section 4, we present the concurrency control mechanisms for multi-stage transactions where we encounter these complications.)

When the final section of the transaction starts, it is anticipated for the final section to observe what the input labels were to the initial section—to know whether the input was erroneous—and what the initial section did—to know what to fix if an error was detected. To avoid adding complexity to the system model and description, we consider that these two tasks are performed by the programmer using database reads and writes. Specifically, the initial section communicates to the final section via writing its input and state to the database.

3.3 Algorithms

Now, we provide the detailed algorithms of Croesus. Parts of the algorithms use a concurrency control component that we present and design in Section 4. We will denote this concurrency control component as CC and a transaction block would either be CC.initial{ } for an initial section and CC.final{ } for a final section. Both transaction blocks get the detected labels as input, but we omit it for brevity.

3.3.1 Client Interface

The client captures frames, gets user input (from auxiliary devices), and displays responses. For example, in a V/AR application, the client captures a frame from the headset camera and sends it to the edge node. Likewise, if there are any associated auxiliary or wearable devices, the client sends the input/commands that correspond to these devices. This process of sending frames and input is continuous—there is no blocking to get the response from the edge node. When a response is received from the edge node, that response is rendered and augmented in the user's view.

3.3.2 Edge Node Algorithms

The edge node is responsible for the initial stage of processing (using the small model M_e), transaction processing, and storage. There are two main components in the edge node: the input processing component and the transaction processing component. The following is a description of the main tasks that are handled by the edge node.

Initialization and Setup. Starting an edge node includes setting up a small model, M_e, a data store ds, and a *transactions bank*. The small model M_e is the one that will be used to process incoming frames. The transaction bank is a data structure that maintains the application transactions and what triggers each transaction. For example, an application may have a transaction t_{bldng} that reads the information about a building that is detected in a frame. The transaction t_{bldng} takes as input the label that is associated with a building. The transactions bank helps the edge node know which transactions should be triggered in response to a label. For example, if a label l_1 represents a label name "Engineering Building" and label l_2 represents a label name "University Shuttle 42", the transaction t_{bldng} should be triggered in response to l_1 but not l_2.

The way the transactions bank helps in making this decision is that it maintains a table, where each row corresponds to a class of labels and the transactions that would be triggered from that class of labels. For example, a row in that table can have a class of labels called "Buildings" and it contains all the labels that would correspond to a building. That row would also have t_{bldng} and any other transactions that should be triggered in response to the "Building" class. A row in the transactions bank may also have other associated triggers, For example, a transaction t_{rsrv} that is used to reserve a study room in a building would be triggered if both a building label is detected in the frame *and* the auxiliary device input is received.

Input and Initial Stage Processing. The initial stage processing represents the input processing using the small model, M_e, in response to a received frame or user input. When a frame f is received by the edge node, it is supplied to the small model M_e. The model M_e returns a set of labels L_e^f. Each label, $L_e^f[i]$, consists of the the name of the label, $L_e^f[i].name$, the confidence of the label, $L_e^f[i].confidence$, and the coordinates of the label, $L_e^f[i].coordinates$. The input processing component removes any labels from the set L_e^f that have low confidence (the threshold for a low confidence is a configuration parameter.) Finally, the input processing component gets the information of all the transactions that correspond to the detected labels, L_e^f, by reading from the transactions bank. The set of triggered transaction, t_f, is sent to the transaction processing component.

Similar to how frames trigger transaction, when a different input is received by the input processing component—such as a click on the auxiliary device—the input processing component generates the set of transactions t_e that corresponds to the input. An auxiliary input might lead to an action that is independent from the captured frame. For example, a click on the menu button may display the menu and general user information. In this case, the entry in the transactions bank is only specified by the input type. Alternatively, the input might be coupled with a specific label class to trigger a transaction. For example, a click would display a captured building's information using t_{rsrv}. In such a case, t_{rsrv} would only be triggered if both the click and a building label are detected. To facilitate such actions, the input processing component matches a received auxiliary input with the labels from the most recently detected labels.

After transactions, t_f, are sent to the transaction processing component (TPC), the frame f is sent to the cloud node to be processed using the cloud model, M_c. This concludes the tasks performed for input processing.

Initial Transaction Section. When the input processing component generates the set t_f for a frame f, these transactions are sent to the TPC. The TPC then triggers the initial section of these transactions. The read and write operations to the database are managed by the concurrency control component by wrapping them in the CC.initial{ } block. (The implementation details of the concurrency control component are presented in Section 4). The initial section of a transaction t would either commit or abort—based on the decision of the concurrency controller. If the initial section aborts, then the abort decision is sent to the client. Otherwise, the response from the initial section is sent to the client, which represents the initial commit point for t. The TPC records the decision for the initial section with the labels, L_e^f, and waits until the corresponding labels are received from the cloud model.

Final Transaction Section. After processing the initial section, the TPC waits for the correct labels, L_c^f, from the cloud node. Once received, the following is performed for each label, $L_e^f[i]$ in L_e^f. The label $L_e^f[i]$ is matched with a label in L_c^f. The matching is performed by finding if the bounding box (represented by the x-y coordinates) of a label in L_c^f overlaps with the bounding box of $L_e^f[i]$. The overlap does not need to be exact—if the label overlap in more than X%, where X is a configuration parameter, then the two labels are considered overlapping. If there are more than two candidates in L_c^f that overlap with $L_e^f[i]$, then the one with the bigger overlap is chosen. There are the following cases of matching the label $L_e^f[i]$ to a label in L_c^f: (1) If an overlapping label cannot be found in L_c^f, then the label $L_e^f[i]$ is considered erroneous and the final section of the corresponding transaction is called with an empty label. (2) If there is a label in L_c^f that overlaps with $L_e^f[i]$ and the label name is the same. In that case, the label $L_e^f[i]$ is considered correct and the final section of the corresponding transaction is called with the same label. (3) If there is a label in L_c^f that overlaps with $L_e^f[i]$ and the label names are different. In that case, the label $L_e^f[i]$ is considered erroneous and the final section of the corresponding transaction is called with the overlapping label from L_c^f.

Once this matching process is complete, then the TPC checks if there are any labels in L_c^f that were not matched. For each one of these labels $L_c^f[i]$, the TPC triggers an initial section and final section with the label in $L_c^f[i]$.

3.3.3 Cloud Node Algorithms

The cloud node has a single task of processing frames using the cloud model, M_c. When a frame f is received from an edge node, the labels, L_c^f, are derived using M_c and then sent back to the edge node.

3.4 Bandwidth Thresholding

A major problem faced by video-analytics applications in the edge-cloud paradigm is the high edge-cloud bandwidth consumption due to the large size of videos. Sending all frames from the edge to the cloud poses a performance challenge due to the communication overhead as well as a monetary overhead due to the cost of transferring data between the edge and the cloud (most public cloud providers charge applications for data communication between the cloud and the Internet). We extend our solution to reduce the reliance on cloud nodes with the goal of overcoming the performance overhead and monetary costs of edge-cloud communication.

The observation we utilize to reduce edge-cloud communication is that we can use the confidence of edge computation to decide whether verifying with the cloud node is necessary. (Confidence here represents the statistical confidence generated by CNN models which is a typical feature of such models.) Specifically, if the confidence of the produced detections in the edge model, M_e, is high, it is likely that the edge model produced correct labels. Therefore, it would not be necessary to send the frame to the cloud. Likewise, if the detections had extremely low confidence, then it is likely that these are erroneous, false detections, and thus sending the frame to the cloud node would be unnecessary as they can be discarded immediately. What is left are detections that have confidence values that are not too high and not too low. These detections are ones that likely indicate the presence of an object of interest, but its label might be incorrect.

More formally, we represent with θ_L and θ_U the lower and the upper confidence thresholds such that $0 \leq \theta_L < \theta_U < 1$. Generally, an object with confidence lower than θ_L is discarded as being likely a false-positive (this is called the *discard interval*). An object with confidence higher than θ_U is assumed to be correct and is not sent to the cloud node (this is called the *keep interval*). Objects with a confidence between θ_L and θ_U are sent to the cloud for validation (this is called the *validate interval*). However, there is a challenge in adopting this model as it is not clear how to derive these confidence thresholds to preserve the integrity of the underlying models. Specifically, a *performance-accuracy trade-off* controls this decision. A large validate interval would lead to better accuracy, since more frames are sent to the cloud for validation and correction. Likewise, a small validate interval would lead to worse accuracy but better performance in terms of average latency and edge-cloud bandwidth utilization. This is complicated further because the size of the validate interval is not the only factor controlling this trade-off. The validate interval size may lead to different performance-accuracy trade-offs based on where it is located in the threshold space from 0–100%.

Optimization Formulation. The input to the optimization problem is a set of video frames $V = \{v_1, \ldots, v_n\}$, and an object query O (e.g., bus), which needs to be detected in the frames. Let n_i be the number of instances of object O detected in frame v_i (by the NN in the edge-node) with confidence $\beta_i = (\beta_i^1, \ldots, \beta_i^{n_i})$ where β_i^k is the confidence corresponding to the k^{th} instance of object O, for $1 \leq k \leq n_i$. We denote this as *edge-confidence*.

Let $m = |\{v_i \in V \mid \exists k \text{ s.t. } \theta_L \leq \beta_i^k \leq \theta_U\}|$ be the number of frames which were sent to the cloud. We define the ratio $\delta(\theta_L, \theta_U) = \frac{m}{n}$ (where n is the number of frames in V) and have the corresponding F-score $f(\theta_L, \theta_U) = \frac{2pr}{p+r}$ where p is precision and r is recall. We want to find (θ_L, θ_U) such that $\delta(\theta_L, \theta_U)$ is minimized and the corresponding $f(\theta_L, \theta_U) \geq \mu$. Let $\mathbb{S} = \{x \in \mathbb{R} \mid 0 \leq x < 1\}$. We have:

$$T = \underset{(x,y)\in\mathbb{S}^2,\mu}{\text{argthresh}}\, f(x,y) := \{(x,y) \in \mathbb{S}^2 \mid f(x,y) \geq \mu\} \tag{1}$$

$$\begin{aligned}(\theta_L, \theta_U) &= \underset{(x,y)\in T}{\text{argmin}}\, \delta(x,y) \\ &:= \{(x^*, y^*) \in T \mid \forall (x,y) \in \mathbb{S}^2, \delta(x^*, y^*) \leq \delta(x,y)\}.\end{aligned} \tag{2}$$

This formulation produces the thresholds (θ_L, θ_U) given μ.

3.5 Generalizing Multi-Stage Processing

In this section, we have focused on models with two stages. This is because the application domain we consider has a two-tier symmetry that invites the use of two sections, one that represents the edge and another that represents the cloud. However, the multi-stage processing model can be utilized for other use cases where the asymmetry has more than two levels. Our designs and treatments can be extended to these cases as we describe in the rest of this section.

Model. In a general multi-stage model, there are m stages, $s_0, \ldots, s_{m-1}$. The first stage, s_0, represents the initial stage of processing and the last stage, s_{m-1}, represents the final stage of processing. All other stages are intermediate stages. The data storage is maintained by the node handling stage s_0. Each stage contains a video/image detection model—where typically the model at stage s_i (denoted m_i) has better detection that model m_j, where $j < i$. A transaction consists of m sections, each one (t_i) corresponding to a stage (s_i).

Processing. When a frame f is received, it is first sent to the initial stage, s_0. The initial stage processes f using m_0 and takes the outcome of the model to process the first section of the transaction t_0. Then, the frame is processed at the next stage s_1—using m_1—and the outcome is used to trigger transaction t_1. This continues until the final stage. If bandwidth thresholding is performed at any stage, then the sequence from initial to final stages might be broken. For example, if at stage s_i, the bandwidth thresholding algorithm (as presented earlier in the section) decides that the frame does not need to be forwarded to the next stage, then the sequence stops and the remaining transaction sections are performed.

4 Multi-Stage Transactions

4.1 Multi-Stage Transaction Model

We consider a new multi-stage transaction model where every transaction comprises of two distinct sections: the initial section and the final section. Each section, s—in a transaction t—consists of read ($r_t^s(x)$) and write ($w_t^s(x)$) operations in addition to control operations to begin (b_t^s) and commit (c_t^s) each section. For example, consider a multi-stage transaction t. The execution of the transaction would look like the following: $b_t^i\ r_t^f(x)\ w_t^i(y)\ c_t^i\ b_t^f\ w_t^f(z)\ c_t^f$ where i stands for the initial section and f stands for the final section.

If the initial section of a transaction commits (called initial commit), then the final section must begin and commit (called final commit) as well. When we say that a transaction t in our model has committed, we mean that both sections of t have committed. Furthermore, the final section of a transaction cannot begin before the initial section. The case for conflicts of transactions also demands special consideration. In our model, we say two transactions to be conflicting if there is at least one conflicting operation in either of the sections. The seemingly simple abstraction of splitting every transaction into two sections complicates the basic notions of the general transaction model. In the following, we take a look at safety and describe two notions of consistency in our model.

4.2 Safety

In the absence of concurrent activity, safety is straight-forward; the initial section is followed by the final section and both are processed as the programmer expects. When concurrency—which is important for performance—is introduced, it challenges the programmer's notion of the sequentiality of running transactions and multi-stage sections (other conflicting transactions may run within and between a transaction's sections.)

For example, consider an application where there are two transactions, t_1 and t_2, each of which increment the value of a data object x by one. Suppose that, for each transaction, the initial stage consists of reading the value of x; the value is increased, and the new value is written in the final section. Therefore, if the two transactions executed concurrently and both t_1 and t_2 read the same value of x, then the final value of x would only increase by one. This is an anomaly because there were two transactions that incremented the value of x and the value of x should have increased by two.

safety is different because it is also actions between sections not only within a transaction. safety here is also different than typical concurrency – it is not about conflicting copies to be merged, it is about a wrong trigger or wrong input. Evidently, multi-stage consistency adds to the complexity involved in traditional consistency guarantees such as serializability in two ways: (1) multi-stage transactions consists of two separate stages. This means that in addition to the concern of concurrent transactions interleaving operations within each section, there is a need to consider whether sections of transactions running *between* the sections of other transactions should be permitted. (2) in multi-stage transactions, inconsistency is not only due to concurrent activity, but also due to erroneous transactions that have an incorrect trigger or input (*e.g.*, an erroneously detected building in the edge stage of processing leads to triggering the wrong transaction and/or supplying it with the wrong input.)

Due to these differences, we revisit transactional consistency in light of multi-stage transactions. We present and discuss two variants of multi-stage transaction consistency. In both variants, we assume that traditional concurrency control mechanisms are used to ensure that each section is serializable relative to other transactions' sections. (This means that each section is atomic and isolated from other sections and that there is a total order on sections.) This leaves the novel challenge to safety that is introduced in our work, which is how these sections can be reordered relative to each other.

4.3 Multi-Stage Serializability (MS-SR)

In MS-SR, we mimic the safety principles of serializability, which is—informally—a guarantee that all transactions execute with the illusion of some serial order of all transactions [22]. When trying to project this to multi-stage transactions, this translates to the requirement that all transactions are processed serially, where the final section of a transaction appears immediately after the initial section. This guarantee can be reduced to serializability by considering that the initial and final sections are part of the same serializable transaction. The main difference is that when the initial section commits, it is a guarantee that the final section would eventually commit—it cannot abort due to unresolved conflicts. As we will see in the rest of this section, this requirements complicates the processing of the initial section.

In order to specify MS-SR formally, we introduce some notations and state our assumptions. We denote with $<_h$, the ordering relation on execution history of transaction sections. This relation represents the ordering relative to the commitment rather than the beginning of the section. For example, $s_a <_h s_b$, denotes that the left-hand side is ordered before the right-hand side, i.e., section s_a is ordered before section s_b.

Consider two conflicting transactions t_k and t_j (i.e., they have at least one conflicting operation in either section), where s_k^i have initially committed before s_j^i initially committed. MS-SR guarantees the following: (1) the final section of the first transaction, s_k^f , must commit after s_k^i. This is the guarantee of multi-stage transactions to commit the initial section before the final section of the transaction. (2) s_k^f must commit before s_j^f. This is due to the MS-SR guarantee that the two sections of the transaction must be ordered next to each other relative to other conflicting transactions. (3) s_k^f must be ordered before s_j^i only if there is a conflict between s_k^f and s_j^i. This is also due to the need to serialize the sections of two conflicting transactions. The condition of the conflict between s_k^f and s_j^i is to capture that if the two sections do not conflict, then they can be reordered in the serializable history. These conditions are represented by the

following formulation, where (a) captures both conditions (1) and (2), and (b) captures condition (3):

$$\text{MS-SR: } (a)\exists t^s \ \Big(s_k^i <_h s_j^i \implies (s_k^i <_h t^s <_h s_j^f \wedge t^s = s_k^f)\Big)$$
$$(b)\text{if conflict in } s_k^f, s_j^i \implies s_k^f <_h s_j^i$$

We elaborate on Example 4.2 to demonstrate the need for MS-SR(a) and MS-SR(b). As an example of MS-SR, consider the two transactions:

$$t_k : b_{t_k}^i r_{t_k}^i(x) c_{t_k}^i b_{t_k}^f w_{t_k}^f(x) c_{t_k}^f \text{ and } t_j : b_{t_j}^i r_{t_j}^i(x) c_{t_j}^i b_{t_j}^f w_{t_j}^f(x) c_{t_j}^f.$$

Further assume that $s_k^i <_h s_j^i$. Condition MS-SR(a) above guarantees that s_k^f is committed after s_k^i and before s_j^f, i.e., we have $s_k^i <_h s_k^f <_h s_j^f$. With MS-SR(a) alone, the following $s_k^i <_h s_j^i <_h s_k^f <_j s_j^f$ is permitted. However, because s_k^f conflicts with s_j^i, then the two sections must be ordered according to MS-SR(b) and the following ordering relations must be met: $s_k^i <_h s_k^f <_h s_j^i <_j s_j^f$. This ordering avoids the anomaly of both transactions reading the same value of x, but one overwriting the value written by the other.

Now, we present a protocol that guarantees MS-SR.

Two Stage 2PL (TSPL): The Two Stage 2PL is the two phase locking protocol [32] modified for our multi-stage transactional model (See Algorithm 1.) Let t_k be a multi-stage transaction comprising of t_k^i and t_k^f. First, the initial section starts executing, locking each accessed data item before reading or writing it. After the initial section finishes processing, the initial commitment cannot be performed immediately. This is because we need to guarantee that the final section can execute and commit as well, due to the requirement of multi-stage transactions. Therefore, the locks of all items that are accessed (or potentially accessed) by the final section must be acquired first. Then, the transaction enters the initial commit phase. Once all the needed input is available for the final section (*e.g.*, the corrected labels from the cloud model), the final section executes, and the transaction enters the final commit phase. Finally, all the locks are released.

```
items ← get_rwsets(t_k^i)
if acquirelocks(items) then
    execute(t_k^i)
    items ← get_rwsets(t_k^f)
    if acquirelocks(items) then
        Initial Commit
        execute(t_k^f)
        Final Commit
    else
        abort
    end
else
    abort
end
releaselocks(get_rwsets(t_k^i))
releaselocks(get_rwsets(t_k^f))
```

Algorithm 1: Two Stage 2PL

Theorem 1. *The TSPL protocol satisfies MS-SR.*

Proof. Consider a pair of conflicting transactions t_p and t_q, where $t_p^i <_h t_q^i$. Following Algorithm 1, each section is serialized relative to each other section because locks are held before execution. Now, we show that the three conditions of MS-SR of ordering sections relative to each other are met. The first guarantee is ordering the initial section before the final section. The algorithm executes the initial section before the final section which guarantees their ordering. The second guarantee that t_p^f is ordered before t_q^f. There is at least one data object o that both t_p and t_q access. Because the final section is only executed after all locks are held for the transaction (including the lock for o), t_p^f would be processed before t_q^f. The third guarantee is that if t_p^f conflicts with t_q^i, then $t_p^f <_h t_q^i$. Assume that the conflict is on data object o. Assume to the contrary that $t_q^i <_h t_p^f$. If that's the case, this means that t_q^i acquired the lock on o before t_p^f and before

the point of initial commitment (because initial commitment only happens after acquiring all locks including the locks for the final section). Because the locks (including the one on o) are not released until t_q finishes, this means that before the lock on o is released, t_q has initially committed. However, t_p initially commits only after acquiring the lock on o, which means that $t_q^i <_h t_p^i$, which is a contradiction to our starting assumption that $t_p^i <_h t_q^i$. □

Discussion. Although MS-SR is an easy-to-use consistency guarantee, it leads to complications and undesirable performance characteristics. The main complication is due to the need to guarantee that committing the initial section would lead to committing the final section. With the stringent requirement that the two sections are serialized so that they appear to be back-to-back in the serialization order, this leads to having to ensure that the locks for the final section can be acquired. The design consequence as we see in the TS-2PL algorithm is that the initial section cannot commit before acquiring the locks of the final section. This leads to one of two consequences: (1) the system can infer what data will be accessed (or potentially accessed) in the final section so that the locks can be acquired and the initial commit happens before having to wait for the cloud model to finish processing, or (2) the transaction would not be able to initially commit until the cloud model returns the correct labels so that it is known what data items are going to be accessed. The first option may require complex analysis or input from the programmer and the second option is prohibitive as it means that the initial section has to wait for a potentially long time, which invalidate the goals of multi-stage transactions. Another complication is that the locks for the initial section must be held until the final section finishes processing which would lead to higher contention.

4.4 Multi-Stage Invariant Confluence with Apologies (MS-IA)

Now, we propose a multi-stage safety criterion that is inspired from invariant confluence [23] and apologies [24]. The initial-final pattern of multi-stage transactions invites the utilization of these concepts as we discuss next.

Guesses and Apologies. The concept of guesses and apologies [24] was introduced to describe a pattern of programming that balances local action versus global action (for example, a local action on a replica versus global action on the state of all replicas in the context of eventual consistency). In this pattern, a *guess* is performed with local information and, then, guesses are reconciled with the global state which would lead to detecting inconsistencies in the local guesses. Such errors lead to *apologies* via undoing actions, administrator intervention, and/or notifications to affected users.

This pattern of guesses and apologies fits our multi-stage edge-cloud transaction model. The initial section represents the guess and the final section represents the apology. To illustrate, consider an example of a multi-player AR game with three players: A with 50 tokens, B with 10 tokens, and C and D with no tokens. The application has a token transfer function transfer(from, to, amount). The initial section performs the transfer, and the final section reconciles any mistakes. Now, assume that the initial section of a transfer t_1 from A to B for 50 tokens took place. Then, the initial section of a transfer t_2 from B to C for 10 tokens took place followed by another transfer t_3 from B to C for 50 tokens. Due to concurrency, assume that the final section of both t_2 and t_3 were performed and that their trigger and inputs were correct. In this case, the final section terminates for both transactions. Then, the final section of t_1 starts. However, the correct input to t_1 turns out to be D instead of B (for example, because the edge CNN model detected player B when it is actually player D as detected by the cloud CNN model.) An apology procedure in the final section could retract the effects of t_1 and any other transactions that depended on it, which are t_2 and t_3.

Using guesses and apologies allows us to process the initial sections of transactions fast while providing a mechanism to overcome the mistakes of the edge best-effort computation. However, it may lead to a cascade of retractions. To overcome this, we propose combining the concept of apologies with invariant confluence as we show next.

Invariant Confluence. In invariant confluence, preserving the application-level invariants is what constitutes a safe execution. In its original form, invariant confluence is intended to reason about transactions mutating the state of different copies of data [23]. Our edge-cloud model is different, involving mutating the state of one (edge) copy. However, an inconsistency might be introduced by the initial section of a transaction with erroneous trigger/input. Our insight is that we can utilize the final (apology) section to act as the *merge* function that attempts to reconcile application-level invariants instead of all potential inconsistencies. In a way, we are flipping the model of invariant confluence systems from a pattern of *check-then-apply* (check if the operation can merge, and decide whether coordination is necessary before doing the operation), to a pattern of *apply-then-check* (do the operation then check whether you can merge, and if you cannot merge, then perform an apology procedure and retract the initial section's effects.)

MS-IA programming pattern. This pattern, when combined with apologies, can lead to reducing the negative consequences of erroneous triggers/inputs. Consider the multi-player AR game application introduced above (when discussing apologies). Assume that the initial sections of t_1, t_2, and t_3, were processed as well as the final sections of t_2 and t_3. At this stage, A, B, and D have no tokens and C has 60 tokens. When the error is discovered, it triggers the final section of t_1. A programmer, equipped with the notions of invariant confluence and apologies, writes the final

section to attempt to perform two tasks: (1) retract the minimum amount of erroneous actions and their effects using apologies, and (2) retain as much state as possible using invariant-preserving merge functions. The specifics of this pattern depends on the application invariants. For example, the final section of the transfer tasks could have the invariant that no player should have less than 0 tokens. The final section of t_1 would retract the 50 tokens that were initially sent to B and sends them to the rightful recipient, player D. This means that B could not have sent a combined 60 tokens to C. The merge function can then decide to retain the 10 tokens sent from B to C, since they are not affected by the error. But, it retracts the 50 tokens. This retraction is accompanied by an apology that depends on the application (*e.g.*, a message is sent to both B and C, with a free game item.)

In terms of the concurrency control guarantee that is needed for MS-IA, the initial section of a transaction must be ordered before its corresponding final section (in addition to our earlier assumption that each section is serialized relative to other transactions' sections). Formally, for an initial section, s_k^i, the following is true:

$$\text{MS-IA: } \exists t^s \ \left(s_k^i <_h t^s \wedge t^s = s_k^f \right)$$

items ← get_rwsets(t_k^i);
if *acquirelocks(items)* **then**
  execute(t_k^i)
end
Initial Commit
releaselocks(get_rwsets(t_k^i))
items ← get_rwsets(t_k^f)
if *acquirelocks(items)* **then**
  execute(t_k^f)
else
  abort
end
Final Commit
releaselocks(get_rwsets(t_k^f))

Algorithm 2: MS-IA Algorithm

Concurrency control. The concurrency control algorithm starts by acquiring all the locks for the initial section, then processing the initial section. When the processing of the initial section is done, the locks are released. Then, when the final section is ready to start, the corresponding locks are acquired before processing the final section. Finally, the locks for the final section are released. Note here that unlike the algorithm for MS-SR, we did not hold the locks for the initial section until the end of the final section and we reach the point of initial-commit immediately after processing the initial section without having to wait to lock or coordinate the final section. The reason for this is that the logic for invariant checking and apologies is embedded in the final section and that we do not need to ensure that the initial and final sections of one transaction are serialized next to each other.

Discussion. To have better performance characteristics, MS-IA presents a more complex programming abstraction than MS-SR because it places the burden of coordination (invariance checking, reconciliation, and apologies) on the programmer. In MS-IA, transactions are written as guesses (in the initial section) and apologies (in the final section). Furthermore, apologies are merge functions that aim to reconcile the inconsistencies caused by incorrect triggers or inputs. Given our apply-then-check pattern, it is possible that some operations cannot be merged. In such cases the final section would undo the effects of the initial section—and any transactions dependent on it. We envision that this pattern of multi-stage guesses and apologies can incorporate advances in merge operators that would allow minimizing the need for undoing transactions. For example, programmers may use merge-able operations in the initial sections and delaying other operations to the final section. This can benefit from—and help empower—the literature of conflict-free and compositional data types. These can be adapted to the initial-final pattern by making merge-able parts in the initial section and enabling other types of operations in the final section.

In Validation-based (optimistic) protocols, which operate in the context of a single transaction, before validation, the outcome of the transaction is not returned to the client and is not exposed to other transactions. Applying validation-based protocol as they are in the edge-cloud setting would be prohibitive because it means that a transaction would not commit until the validation step - that would happen after cloud processing - is ready. The MS-IA pattern, on the other hand, divides the transaction logic to two sections each acting as an independent transaction, where the first one

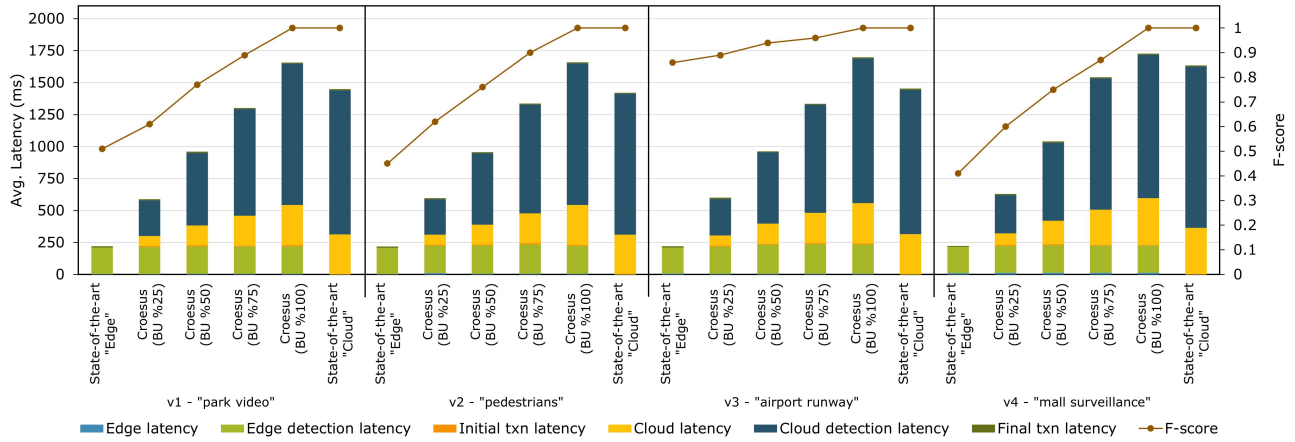


Figure 2: Croesus vs. state of the art baselines: Latency and F-score of running Croesus over four videos. Some values are minute and are hard to show on the figure.

commits before the second section starts, which allows returning responses to clients and exposing the outcome to other transactions (even before the final section and without having to wait for the processing at the cloud).

4.5 Multi-Partition Operations

The transaction processing protocols presented in this section focus on transactions that are local to a partition. In the case of distributed transactions (spanning multiple partitions), the presented algorithms need to be extended. In particular, in the multi-partition case, the data objects that are accessed by a transaction (whether in the initial or final sections) can be in multiple partitions. Locking data objects in remote partitions will be performed by sending the lock requests to the remote edge node that is responsible for the partition. The second difference is that after the transaction finishes, the partitions engage in a two-phase commit protocol to ensure that the distributed commit is performed in an atomic way. This atomic commitment step is performed in the following cases: (1) for MS-SR, it is performed at the end of the final section, (2) for MS-IA, it is performed at the end of both the initial and final sections. The reason for not performing this step at the end of the initial section in MS-SR is that the locks are not released until the end of the corresponding final section.

5 Evaluation

In this section, we show how Croesus manages the trade-off between performance and accuracy of two models with different characteristics: (1) YOLOv3 [27, 33] as the cloud model, which is reported to achieve 45 FPS on high-end hardware and achieves high accuracy. (2) Tiny YOLOv3 [27, 33]—which is a compact version of YOLOv3—for the edge model. Tiny YOLOv3 is faster but less accurate than YOLOv3 [33].

We compare Croesus with two baselines: • State-of-the-art edge baseline: this baseline represents a performance-centric video analytics applications where a compact model (Tiny YOLOv3) is deployed on the edge machine for lower latency. • State-of-the-art cloud baseline: this baseline represents accuracy-centric video analytics applications where a computationally expensive model (YOLOv3) is deployed on a resourceful cloud machine for better accuracy.

5.1 Experimental setup

Our evaluations are performed on Amazon's AWS EC2 services. Edge machines are implemented on either t3a.xlarge instances (for the default setups) and t3a.small (for experiments with limited resources). t3a.small machines have 2 virtual CPUs and 2GiB of memory and t3a.xlarge machines have 4 virtual CPUs and 16GiB of memory. Machine locations are either in California or Virginia. The default setup is of an edge machine in California and a cloud machine in Virginia.We implement a prototype of Croesus in Python. In addition to model detection, the edge node maintains a data store and processes transactions according to the MS-IA algorithm. Transactions are constructed by randomly selecting keys to read or write to the database in response to detected labels.

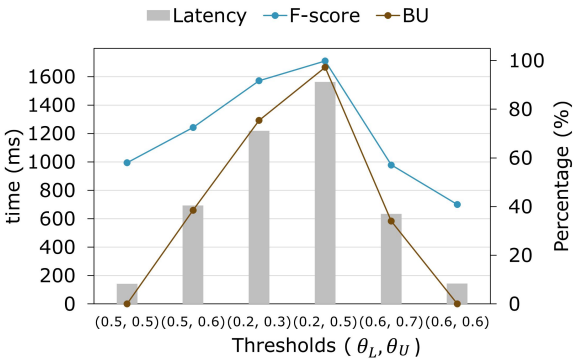


Figure 3: Croesus latency vs. accuracy for different pairs of thresholds

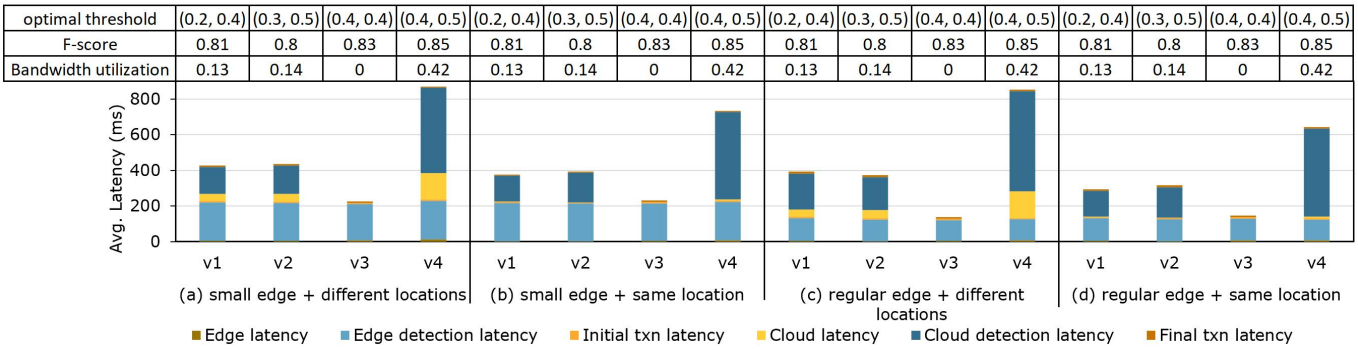


Figure 4: Latency in different setups for the optimal case that was dynamically configured by Croesus.

We evaluate accuracy and performance as follows: Accuracy is measured as the F-score. Performance is measured in two ways: (1) Latency, which we define as the time required to commit transactions in the system. (2) Edge-Cloud Bandwidth Utilization (BU), which we define as the ratio of frames being sent to the cloud relative to all processed frames. This metric is proportional to the number of corrections that need to be made in the final transaction. We consider the YOLOv3 output to be the ground truth and we use it to compare Creosus' results and calculate the F-score. When the overlap between the truth boundaries and the predicted boundaries is more than %10, we consider the prediction correct. The calculation of the F-Score does not depend on the percentage of frames that are sent or not sent to the cloud, but rather on the accuracy of the detection from the perspective of the client (i.e., the accuracy of the detection *and* apologies, if any.) There is, however, a correlation between sending more frames to the cloud as it means that more errors are corrected by the more accurate cloud model.

Experiments run on a subset of five types of videos: Street traffic (vehicles), street traffic (pedestrians), mall surveillance (all three querying for 'person'), airport runway querying for 'airplane', and home video of pet in the park querying for 'dog'. Each detection acquired for each frame triggers a transaction that has 6 operations, half of these mutate the state

Table 1: Comparison between state-of-the-art edge and cloud and optimal threshold Croesus

	Accuracy			Latency (ms)		
	Croesus	Edge	Cloud	Croesus	Edge	Cloud
v1	0.81x	0.5x	1	427.02 (226.16)	210.74	1452.5
v2	0.8x	0.45x	1	434.81 (224.41)	207.97	1427.69
v3	0.83x	0.86x	1	225.63 (218.17)	211.19	1455.66
v4	0.85x	0.41x	1	863.96 (235.02)	214.65	1638.89

Table 2: The effect of the cloud model size.

Croesus cloud model	Optimal threshold	Croesus F-score	Bandwidth Utilization	Detection latency (sec)
YOLOv3-320	(0.2, 0.3)	0.84	0.61	0.70
YOLOv3-416	(0.4, 0.5)	0.86	0.44	1.12
YOLOv3-608	(0.4, 0.6)	0.83	0.58	2.34

of the database by inserting data items, and the other half read from previously added items. This mimics a write-heavy workload of YCSB (Workload A) [34]. Unless we mention otherwise, we use MS-IA as the consistency guarantee.

5.2 Experimental results

5.2.1 Performance vs. accuracy trade-off

Figure 2 shows the trade-off between the latency and accuracy as BU varies on four videos: park video (v1), street traffic (v2), airport runway (v3) and mall surveillance (v4). For each video, we compare different BU configurations with the state-of-the-art edge and cloud solutions. In the figure, the stacked bars represent the latency breakdown for each experiment. Edge latency and cloud latency represent the average time needed to send a frame to the edge and to the cloud, respectively. The edge detection latency and cloud detection latency are defined as the average time it takes the tiny YOLOv3 and YOLOv3 models, respectively, to produce the detected objects list in a frame. The initial transaction and final transaction latency are very minute and hard to show in the figure, but they represent the time it takes to commit a transaction after detection is done. The F-score metric is shown as a marked line.

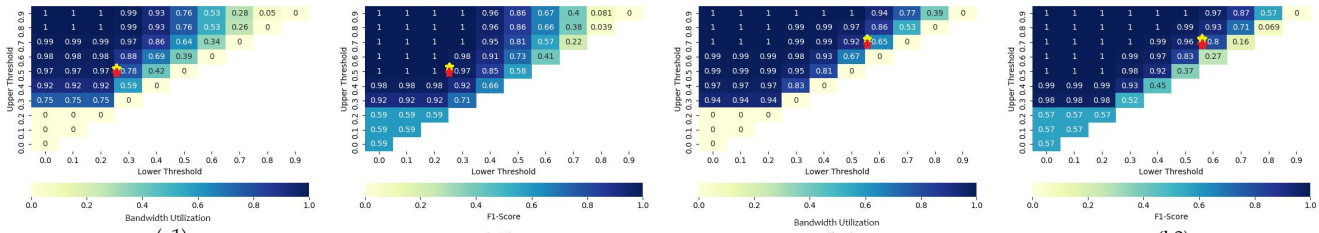
As shown in Figure 2, Croesus processes transaction updates in the initial phase (measured by edge latency and edge detection latency), up to $6.9\times$ faster than the case with full BU while maintaining high accuracy (F-score up to $\%94$ in the case of "airport runway") by utilizing the cloud corrections and final transaction. The client observes two latencies: the first is the real-time initial processing at the edge which corresponds to edge latency, edge detection latency, and initial transactions latency. The second is for the final processing after corrections, if any, from the cloud, which corresponds to all the latency types shown in the figure. As BU increases, the amount of frames sent to the cloud, and consequently the average cloud-related latencies, increases. When BU is 100%, the total cloud latency for Croesus becomes even higher than state-of-the-art cloud because it incurs all the overheads of the state-of-the-art cloud in addition to the overhead of Croesus methods.

The trend of increasing Croesus cloud latency as BU increases is observed in videos 1, 2, and 4. However, a unique trend appears for video 3 (querying for 'airplane' on the airport runway video). In this video, the state-of-the-art edge produce high accuracy due to the nature of the video (an object that is detected by the edge model with high confidence). This asserts the need for dynamic optimization over the detection thresholds for different applications in order to address workload differences. Croesus' dynamic optimization ensures the best balance of the trade-off between accuracy and latency depending on the needs of each application.

Figure 3 demonstrates the effect of choosing different thresholds on the latency in Croesus. We demonstrate the results using the street traffic video querying for vehicles. It shows the total Croesus cloud latency and the BU percentage as the threshold pairs for detections are varied. For example, a threshold pair (0.5, 0.6) means that only detections with confidence values in the edge mode that are within these two values are sent to the cloud for verification. Detections with lower confidence values are discarded and ones with higher confidence values are assumed correct by the edge node and are not verified (however, erroneous detections are still accounted for in the F-score.)

When the thresholds are set to (0.5, 0.5) the resulting BU is $\%0$ since no frames will be sent to the cloud for validation. The resulting accuracy is comparable to the edge only baseline at $\%58$. For a threshold pair of (0.5, 0.6), the latency increases due to more results being validated in the cloud. The resulting BU is $\%38.5$ while the F-score increases by $\%25$. When the BU reaches $\%97.2$, the accuracy reaches $\%99.8$. For thresholds (0.6,0.7), the BU is only $\%4$ lower than the BU of the thresholds (0.5, 0.6). However, the F-score decreases by more than $\%21.24$. This shows that although two pairs may have similar BU values, their corresponding F-score can be significantly different. It indicates the importance of dynamically optimizing for an optimal pair of thresholds that balance the trade-off between the latency and accuracy while prioritizing thresholds that yield higher accuracy.

Another observation from Figure 3 is that the rate at which the bandwidth utilization increases is faster than the rate of F-score increase over different threshold pairs. This is an indicator that increasing dependence on the cloud does not necessarily improve accuracy dramatically.



(a1) (a2) (b1) (b2)

Figure 5: Croesus bandwidth utilization vs. accuracy based on the threshold pair choice. a) traffic video querying "person" ($\mu = 0.90$) and b) mall surveillance querying "person" ($\mu = 0.80$). For all pairs of lower threshold (θ_L) and upper threshold (θ_U). Dynamically chosen pair: yellow star using brute force, red star using gradient step.

The effect of changing the cloud model size in Croesus is demonstrated in Table 2. In this experiment, we set $\mu = 0.8$ and compare the performance of Croesus while using three different cloud model sizes: YOLOv3-320, YOLOv3-416, YOLOv3-608, where the number at the end of each model's name represents the width and height used in the neural network model. Therefore, a larger number indicates a larger model. As the cloud model size gets larger, the detection latency gets larger as well. This is the main impact of utilizing different model sizes. The different models have different accuracy characteristics as well. However, using them in the Croesus framework does not demonstrate such differences in the resulting F-score and BU. This is because the optimal thresholds are set based on the used cloud model to achieve the desired minimum accuracy, μ.

5.2.2 Optimal threshold performance on different setups

Figure 4 shows the accuracy and performance results of Croesus for different videos when using the optimal threshold. These experiments run across four different setups: (a) Small edge, different locations: Edge machines are of type t3a.small while cloud machines are of type t3a.xlarge. Edge machine are located in California and cloud machines are in Virginia. (b) small edge, same location: Small edge, different locations: Edge machines are of type t3a.small while cloud machines are of type t3a.xlarge.Edge and cloud machines are physically located in the same location. (c) Regular edge, different location: Edge and cloud machines are both of type t3a.xlarge. Edge machine are located in California and cloud machines are in Virginia. (d) Regular edge, same location: Edge and cloud machines are physically located in the same location and are both of type t3a.xlarge.

This figure demonstrates the improvement in latency that the optimal thresholds provide compared with the performance shown in Figure 2 (For a clearer presentation, we show the comparison numbers in Table 1, where the number inside the parentheses in Croesus is the latency of the initial transaction.). Also, it shows the effect of resource allocation and geographical location on performance, and the importance of dynamic threshold optimization to address the differences in applications.

In the case of applying the optimal thresholds, we see improvement in the final latency over the state-of-the-art cloud implementation by up to $\%85$ (but as low as $\%47$ for the case of v4). In addition, committing the initial transaction is always comparable to the state-of-the-art edge solutions. Even though the final transaction in Croesus can take up to $\%75$ more than the edge only implementation, the accuracy improvements is significant and can justify the slight delay after the initial transaction.

In addition, the F-score of optimal Croesus is 2.1x higher than the F-score of edge-only in video v4. In the case of video v3, the accuracy is comparable to the state-of-the-art accuracy because the optimal thresholds represent a near $\%0$ bandwidth utilization. This is possible in application where objects are expected to be easier to detect in each frame. The figure also shows that as the geographical distance between the edge and the cloud decreases (when placed in the same location), Croesus performance improves. In addition, the performance improves when edge resources are maximized.

5.2.3 Dynamic preprocessing optimization

Figure 5 shows the bandwidth utilization and accuracy as we vary the optimization thresholds (the lower threshold θ_L and upper threshold θ_U). The heatmaps illustrate the gradual shift in the balance between bandwidth utilization and accuracy.

bandwidth utilization/accuracy trade-off. Figure 5(a1) for BU and Figure 5(a2) for F1-Score show the trend where increasing the lower threshold and the gap between the two optimization thresholds results in a higher throughput. For

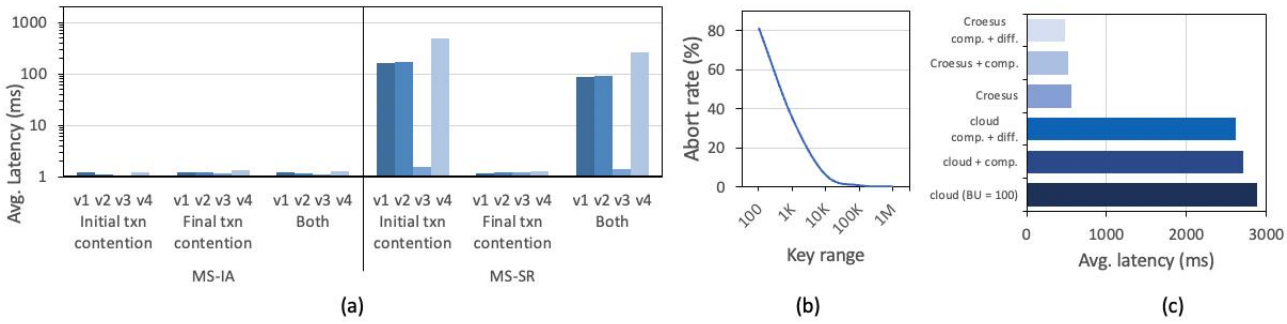


Figure 6: (a) Comparing lock contention of MS-SR and MS-IA measured as the average latency of holding locks. (b) Abort rate of MS-SR transactions. (c) Hybrid system techniques.

example, when the optimization pair is (0.2, 0.4), the F-score is $\%98$ since this pair of thresholds result in a high BU at$\%92$. However, when the optimization pair is (0.3, 0.4) the bandwidth utilization drops to $\%59$ while the F-score remains relatively high $\%92$. We are able to conserve the edge-cloud communication by more than %35.9 while maintaining relatively high accuracy.

Figures 5(b1) for BU and 5(b2) for F1-Score show the same trends as the previous set of heatmaps. However, we notice a sudden jump in bandwidth utilization and F-score results. This is due to the quality of this second video where objects are smaller and not as clear as the first video. In this case, utilizing edge-cloud communication increases the quality of detections dramatically compared to edge detections. For example, for the optimization pair (0.4,0.5) %81 of frames are sent to the cloud and the F-score is %92. However, when the optimization pair is (0.4,0.4) no frames are sent to the cloud and the F-score decreases to %45.

Dynamically finding the optimal solution. We implemented two approaches to acquire the optimized pair of thresholds. The first is a brute force method that evaluates the whole space of threshold pairs. In it, we obtain the optimal pair for balancing the trade-off (shown as a yellow star). The second approach uses a gradient step with our optimization formulation. Using gradient step is 2.2x times faster (shown as a red star). In both cases, bandwidth utilization is $< \%78$, accuracy is at least $\%49$ higher than an edge model.

5.2.4 Comparing MS-SR and MS-IA

In the next set of experiments, we measure the performance differences between the two proposed consistency levels: MS-SR and MS-IA. (In this set of experiments we use video v4 with the query "person".) The main difference between the two consistency levels is that the locks in the initial section of MS-SR are held until the end of the whole transaction, whereas in MS-IA, the locks are released after the initial section. This results in increasing the lock contention in MS-SR. Figure 6(a) shows the difference in contention by measuring the average time locks are held in MS-SR and MS-IA (denoted average latency in the figure.) While the average latency of MS-IA is in the order of milliseconds, the average latency of initial sections in MS-SR is in the order of hundreds of milliseconds. This is because the locks are not released until the final section is performed which means that the locks are held while the frame is being processed using the cloud model which takes a significant amount of time.

The contention difference leads to a high likelihood of aborts in MS-SR. Figure 6(b) shows the abort rate of transactions in MS-SR while emulating a high contention scenario of hot sports with different sizes. The x-axis (key range) is the key range of the hot spot that the transactions are trying to access. In this model, transactions are executed in batches of 50 transactions per batch where each transaction has 5 update operations. The figure shows that the abort rate can be significant when the hot spot has a size that is less than 10K keys. This demonstrates the benefit of using MS-IA to overcome the hot spot contention problems while using MS-SR. The figure does not show the abort rate of MS-IA transactions as the rate is 0% for all cases. This is because our implementation uses a single-threaded sequencer to order transactions in batches so that conflicting transactions do not overlap. This is possible as the transactions do not have to hold locks for prolonged durations.

5.2.5 Hybrid edge-cloud techniques

Hybrid edge-cloud techniques have been proposed to process object detection models [1, 35–37]. These techniques generally work by performing some pre-processing steps at the edge node before sending the frame to be detected at the

cloud. We compare with two such techniques that were utilized in various forms in prior work [1,36]: (1) *compression* in which the frame is compressed before sending it to reduce the communication bandwidth and latency, and (2) *difference communication* in which only the difference between the current frame and a reference frame is sent to the cloud. These techniques, if implemented in isolation, would achieve a small improvement over the performance of the state-of-the-art cloud baseline that we compared with as they would still require sending all frames for detection in the cloud. We show this in the evaluations on the park video v1 with the larger cloud model (YOLOv3-608) in Figure 6(c) under cloud+compression and cloud+compression+difference. These evaluations apply the hybrid techniques which improves the latency as less data need to be sent. However, this is a small improvement because the latency is dominated by the detection latency at the cloud.

An alternative view of these techniques is as methods to augment with edge-cloud Croesus. Figure 6(c) also shows how augmenting compression can improve the final commit latency in Croesus (under Croesus+compression and Croesus+compression+difference). The improvement is small because the model detection latency in the cloud is the dominant latency (as we show in previous evaluations.)

6 Related Work

The requirement of real-time processing has been tackled by real-time Databases (RTDB) [38] that aim to process data in predictable short time. Our method differs by allowing to manage the trade-off of performance and accuracy and providing the illusion of both a fast and accurate processing. A hybrid edge-cloud model (and similar caching-based models) have recently been used [1,35–37] to take advantage of cloud computing to process data on neural networks, as well as leveraging resources at the edge. Our work extends these efforts by providing a multi-stage transactional model that enables programmers to reason about this hybrid edge-cloud model. In particular, these hybrid edge-cloud models can be augmented with the edge-cloud model of Croesus to improve the edge-to-cloud latency. However, when hybrid edge-cloud models are used in isolation, they would incur the high costs of edge-to-cloud communication for all frames since they require performing the detection in the cloud.

The multi-stage transaction model differs from existing abstractions in that each transaction is split into two asymmetrical sections. This makes traditional consistency models [22] unsuitable for multi-stage transactions. The pattern of initial-final sections resemble work on eventual consistency [39] and Transaction Chains [40] but differs in one main way: the inconsistencies in the multi-stage model are external to the database. They are caused by erroneous inputs or triggers. In eventual consistency and Transaction Chains, inconsistency is caused by concurrent operation across different copies. This leads to similarities and differences, which led us to adapt prior relevant literature. Multi-stage transactions resemble work on long-lived transactions (LLT) as well, such as Sagas [41]. Multi-stage transactions can be viewed as a special case of LLT's—with a transaction and a follow-up correction/compensation transaction—which enables simpler and more efficient solutions.

We view Croesus as a data layer solution that builds on top of asymmetric environments which - like edge-cloud - may include the lambda architecture [42] with both batch processing (slower but more accurate) and speed/real-time processing (faster but less-accurate). The contributions of Croesus can be applied to the lambda environment [43] by using multi-stage transactions (where the initial section is processed after real-time processing and the final section is processed after batch processing), and thus provide Croesus benefits to lambda programmers.

7 Conclusion

We presented Croesus, a multi-stage processing system for video analytics and a multi-stage transaction model which optimizes the trade-off between performance and accuracy. We present two variants of transnational consistency for multi-stage transactions—multi-stage serializability and multi-stage invariant confluence with apologies. Our evaluation demonstrates that multi-stage processing is capable of managing the accuracy-performance trade-off and that this model provides both immediate real-time responses and high accuracy.

Although we have presented the concept of multi-stage processing and transactions in the context of edge-cloud video analytics and processing [44–48], these concepts are relevant to many problems that share the pattern of needing immediate response and complex processing. Our future work explores these applications. One area of future work is to apply this pattern of multi-stage processing to blockchain systems with off-chain components [49–51]. In such a case, the first stage is performed in the off-chain component while the final stage is performed after validation from the blockchain. Another area we plan to explore is to integrate the multi-stage processing structure with global-scale edge placement and reconfiguration [52,53]. This will allow utilizing multi-stage processing more efficiently by controlling where the stages are performed and what edge/cloud datacenters to utilize.

8 Acknowledgement

This research is supported in part by the NSF under grant CNS-1815212.

References

[1] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: optimizing neural network queries over video at scale," *arXiv preprint arXiv:1703.02529*, 2017.

[2] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[3] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.

[4] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," *arXiv preprint arXiv:1908.09791*, 2019.

[5] P. Lincoln *et al.*, "From motion to photons in 80 microseconds: Towards minimal latency for virtual and augmented reality," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 4, pp. 1367–1376, 2016.

[6] S. Chen *et al.*, "Vehicle-to-everything (v2x) services supported by lte-based systems and 5g," *IEEE Communications Standards Magazine*, vol. 1, no. 2, pp. 70–76, 2017.

[7] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arxiv 2015," *arXiv preprint arXiv:1510.00149*, 2019.

[8] H. Kim, M. U. K. Khan, and C.-M. Kyung, "Efficient neural network compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 12 569–12 577.

[9] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

[10] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression," *arXiv preprint arXiv:1702.04008*, 2017.

[11] C. Chen, F. Tung, N. Vedula, and G. Mori, "Constraint-aware deep neural network compression," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 400–415.

[12] Y. Xu, Y. Wang, A. Zhou, W. Lin, and H. Xiong, "Deep neural network compression with single and multiple level quantization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[13] Y. Choi, M. El-Khamy, and J. Lee, "Universal deep neural network compression," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 715–726, 2020.

[14] A. Dubey, M. Chatterjee, and N. Ahuja, "Coreset-based neural network compression," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 454–470.

[15] T. Chen, H. Liu, Q. Shen, T. Yue, X. Cao, and Z. Ma, "Deepcoder: A deep neural network based video compression," in *2017 IEEE Visual Communications and Image Processing (VCIP)*, 2017, pp. 1–4.

[16] Z. Liu, T. Liu, W. Wen, L. Jiang, J. Xu, Y. Wang, and G. Quan, "Deepn-jpeg: A deep neural network favorable jpeg-based image compression framework," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3195970.3196022

[17] Y. Li, D. Liu, H. Li, L. Li, Z. Li, and F. Wu, "Learning a convolutional neural network for image compact-resolution," *IEEE Transactions on Image Processing*, vol. 28, no. 3, pp. 1092–1107, 2019.

[18] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019.

[19] D. Racki, D. Tomazevic, and D. Skocaj, "A compact convolutional neural network for textured surface anomaly detection," in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018, pp. 1331–1339.

[20] V. J. Lawhern, A. J. Solon, N. R. Waytowich, S. M. Gordon, C. P. Hung, and B. J. Lance, "Eegnet: a compact convolutional neural network for eeg-based brain–computer interfaces," *Journal of neural engineering*, vol. 15, no. 5, p. 056013, 2018.

[21] Y. Guo, Z. Yang, K. Liu, Y. Zhang, and W. Feng, "A compact and optimized neural network approach for battery state-of-charge estimation of energy storage system," *Energy*, vol. 219, p. 119529, 2021.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[23] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 185–196, 2014.

[24] P. Helland and D. Campbell, "Building on quicksand," in *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. www.cidrdb.org, 2009. [Online]. Available: http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf

[25] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5455–5516, 2020.

[26] V. A. Sindagi and V. M. Patel, "A survey of recent advances in cnn-based single image crowd counting and density estimation," *Pattern Recognition Letters*, vol. 107, pp. 3–16, 2018.

[27] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[28] S. Wen, H. Wei, Z. Yan, Z. Guo, Y. Yang, T. Huang, and Y. Chen, "Memristor-based design of sparse compact convolutional neural network," *IEEE Transactions on Network Science and Engineering*, vol. 7, no. 3, pp. 1431–1440, 2019.

[29] K. Zhang, J. Chen, T. Zhang, and Z. Zhou, "A compact convolutional neural network augmented with multiscale feature extraction of acquired monitoring data for mechanical intelligent fault diagnosis," *Journal of Manufacturing Systems*, vol. 55, pp. 273–284, 2020.

[30] D. Racki, D. Tomazevic, and D. Skocaj, "A compact convolutional neural network for textured surface anomaly detection," in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2018, pp. 1331–1339.

[31] Z. Xu and R. C. Cheung, "Accurate and compact convolutional neural networks with trained binarization," *arXiv preprint arXiv:1909.11366*, 2019.

[32] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[33] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[35] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 155–168. [Online]. Available: https://doi.org/10.1145/2809695.2809711

[36] P. M. Grulich and F. Nawab, "Collaborative edge and cloud neural networks for real-time video processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 2046–2049, 2018.

[37] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[38] S. H. Son, R. David, and B. Thuraisingham, "Improving timeliness in real-time secure database systems," *ACM SIGMOD Record*, vol. 25, no. 1, pp. 29–33, 1996.

[39] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Queue*, vol. 11, no. 3, pp. 20–32, 2013.

[40] Y. Zhang *et al.*, "Transaction chains: achieving serializability with low latency in geo-distributed storage systems," in *SOSP*, 2013.

[41] H. Garcia-Molina and K. Salem, "Sagas," *SIGMOD Rec.*, vol. 16, no. 3, p. 249–259, Dec. 1987. [Online]. Available: https://doi.org/10.1145/38714.38742

[42] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 2785–2792.

[43] J. Warren and N. Marz, *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.

[44] F. Nawab, "Wedgechain: A trusted edge-cloud store with asynchronous (lazy) trust," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 408–419.

[45] F. Nawab, D. Agrawal, and A. El Abbadi, "Dpaxos: Managing data closer to users for low-latency and mobile applications," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1221–1236.

[46] ——, "Nomadic datacenters at the network edge: Data management challenges for the cloud with mobile infrastructure." in *EDBT*, 2018, pp. 497–500.

[47] S. Gazzaz and F. Nawab, "Collaborative edge-cloud and edge-edge video analytics," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 484–484.

[48] N. Mittal and F. Nawab, "Coolsm: Distributed and cooperative indexing across edge and cloud machines," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 420–431.

[49] D. Abadi, O. Arden, F. Nawab, and M. Shadmon, "Anylog: a grand unification of the internet of things," in *Conference on Innovative Data Systems Research (CIDR '20)*, 2020.

[50] M. Alaslani, F. Nawab, and B. Shihada, "Blockchain in iot systems: End-to-end delay evaluation," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8332–8344, 2019.

[51] F. Nawab and M. Sadoghi, "Blockplane: A global-scale byzantizing middleware," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 124–135.

[52] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi, "Global-scale placement of transactional data stores." in *EDBT*, 2018, pp. 385–396.

[53] ——, "Db-risk: The game of global database placement," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 2185–2188.