Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines

Francisco Romero*
faromero@stanford.edu
Stanford University

Mark Zhao*
myzhao@stanford.edu
Stanford University

Neeraja J. Yadwadkar
neeraja@cs.stanford.edu
Stanford University

Christos Kozyrakis
christos@cs.stanford.edu
Stanford University

ABSTRACT

The proliferation of camera-enabled devices and large video repositories has led to a diverse set of video analytics applications. These applications rely on video pipelines, represented as DAGs of operations, to transform videos, process extracted metadata, and answer questions like, "Is this intersection congested?" The latency and resource efficiency of pipelines can be optimized using configurable knobs for each operation (e.g., sampling rate, batch size, or type of hardware used). However, determining efficient configurations is challenging because (a) the configuration search space is exponentially large, and (b) the optimal configuration depends on users' desired latency and cost targets, (c) input video contents may exercise different paths in the DAG and produce a variable amount intermediate results. Existing video analytics and processing systems leave it to the users to manually configure operations and select hardware resources.

We present Llama: a heterogeneous and serverless framework for auto-tuning video pipelines. Given an end-to-end latency target, Llama optimizes for cost efficiency by (a) calculating a latency target for each operation invocation, and (b) dynamically running a cost-based optimizer to assign configurations across heterogeneous hardware that best meet the calculated per-invocation latency target. This makes the problem of auto-tuning large video pipelines tractable and allows us to handle input-dependent behavior, conditional branches in the DAG, and execution variability. We describe the algorithms in Llama and evaluate it on a cloud platform using serverless CPU and GPU resources. We show that compared to state-of-the-art cluster and serverless video analytics and processing systems, Llama achieves 7.8× lower latency and 16× cost reduction on average.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **Client-server architectures**.

KEYWORDS

scheduling, heterogeneous, distributed systems, serverless computing, video analytics

ACM Reference Format:
Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3472883.3486972

1 INTRODUCTION

Video traffic is exploding in scale, predicted to account for over 82% of all internet traffic by 2022 [6]. A myriad of domains use *video pipelines*, with tens of video analytics and processing operations, to extract meaningful information from raw videos. For example, an AMBER Alert application can leverage traffic cameras across a city to pinpoint specific individuals and cars [7]. To do so, the application uses a pipeline to first detect frames with people and/or cars, and then match them to specific individuals' faces and car descriptions, respectively. As video analytics research continues to flourish, we expect a perpetual proliferation of emerging domains that depend on video pipelines, such as smart cities [20, 42, 45, 78], surveillance analytics [23], healthcare [44], and retail [19].

*Denotes equal contribution.

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486972

The pervasive use of video analytics applications has led to significant challenges. Video pipelines must meet a wide range of latency, throughput, and cost targets to be practical across applications. For example, a pipeline to detect cars and people in a traffic feed should be tuned to be more cost efficient for city traffic planners with relaxed latency targets, while the same pipeline must be tuned to meet strict latency targets for AMBER Alert responders [7]. Video analytics and processing frameworks must tune pipeline operation knobs (e.g., sampling rate, batch size, hardware target, and resource allocation) to meet the unique latency or cost requirements of diverse applications. However, automatically tuning these knobs is difficult for the following reasons.

Operations exhibit performance variation across heterogeneous hardware. Hardware accelerators (e.g., GPUs [27], FPGAs [37, 70], TPUs [47], and vision chips [2]) provide significant performance benefits for many pipeline operations. Tuning knobs across these heterogeneous accelerators can have a huge impact in the performance and efficiency of video pipelines. We observed a 3.7× latency variation by tuning CPU cores, GPU memory, and batch size for operations in a representative AMBER Alert pipeline processed using Scanner [61]. While recent research has proposed mechanisms to tune operation knobs based on resource usage [42, 45, 73], they are limited to simple pipelines and homogeneous hardware platforms. Furthermore, they rely on hours to days of profiling for *each* new pipeline, video, and latency target [18, 78].

Pipelines can have input-dependent execution flow. An input video's contents influence the execution flow of a pipeline in two ways. First, the number of intermediate outputs for an operation may depend on the frame being processed. For the AMBER Alert pipeline, the object detector operation will output a variable number of cropped people and/or car images to be processed by subsequent operations. Second, downstream operations may be conditionally executed based on the intermediate output. For example, if there are only people in a frame, no car classification is needed. Consequently, tuning configuration knobs and resource allocations dynamically based on video content is critical for performance and efficiency. We found the static configurations made by gg [35], a general purpose serverless framework, degraded performance by as much as 57% for the AMBER Alert pipeline. Some systems, such as VideoStorm [78] and GrandSLAm [48], only support simple sequential pipelines with deterministic flow.

Systems that use serverless platforms as their backend (e.g., ExCamera [36], gg [35], PyWren [46], and Sprocket [22]) execute applications by using thousands of short-lived functions [3, 5, 9]. The function-level resource allocation offered by serverless platforms makes them an attractive option for processing video pipelines, as they enable dynamic tuning for each operation invoked. However, existing serverless offerings lack support for heterogeneous hardware accelerators and application constraints such as latency targets. Users must still manually, and perhaps exhaustively, explore operation knobs and resource allocation options.

We present Llama, a video analytics and processing framework that supports heterogenous hardware and automatically tunes each operation invocation to meet diverse latency targets for the overall pipeline. Llama is a full-fledged serverless framework that provides a serverless experience to its users: fine-grained billing and does not require users to express the resources or operation knob configurations needed to meet their latency targets.

Llama is divided into two parts: an offline specification phase, and an online optimization phase. The offline specification phase allows users to specify their pipeline, and performs a one-time, per-operation profiling that allows Llama to automatically tune operation invocations as the pipeline runs. Unlike existing systems [28], this profiling does not need to be repeated as the pipeline or input video changes.

During the online phase, Llama leverages two key ideas to meet diverse latency targets. First, Llama *dynamically* computes how much time can be spent on each invocation to meet the pipeline latency target (i.e., per-invocation latency targets). By computing a per-invocation latency target, Llama can dynamically explore the configuration space for each invocation and adapt to performance volatility and input-dependent execution flows. Second, Llama dynamically runs a cost-based optimizer that determines the most efficient operation configuration that meets the per-invocation target. To do so, Llama (a) uses *early speculation and late commit*: a technique for choosing an initial operation knob configuration during pipeline processing, and revisiting the configuration right before execution, (b) leverages *priority-based commit* to prioritize operations based on hardware affinity and DAG dependencies, and (c) employs *safe delayed batching* to batch operations for efficiency as long as doing so does not violate per-invocation targets.

We deploy Llama on Google Cloud Platform with serverless CPU and GPU backends and evaluate its efficiency and ability to meet latency targets for five video analytics and processing pipelines. By dynamically configuring operations for both CPU and GPU based on pipeline latency targets, Llama achieves on average 7.8× latency improvement and 16× cost reduction compared to four state-of-the-art cluster and serverless video analytics and processing systems: Nexus [63], Scanner [61], gg [35], and GrandSLAm [48].

		InferLine [28]	GrandSLAm [48]	VideoStorm [78]	Focus [42]	Nexus [63]	Scanner [61]	gg [35]	Sprocket [22]	LLAMA
Features	Performance targets	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes
	General operations	No	No	Yes	No	No	Yes	Yes	No	Yes
Challenges	Traverse large configuration space	Limited¶	Limited¶	Limited†	Limited¶	Limited¶	No	No	No	Yes
	Handle input-dependent exec. flow	No	No	No	No	Yes	No	Limited‡	No	Yes
	Dynamically adjust resource alloc.	Yes	No	Limited§	No	Limited§	No	Yes	Yes	Yes

Table 1: Comparison of existing video processing systems with LLAMA based on whether they (a) support performance targets and general operations, and (b) address the challenges of meeting performance targets for general video pipelines. ¶Limited to domain-specific knobs. †Large profiling overhead. ‡Cannot handle branching. §Limited to single hardware platform.

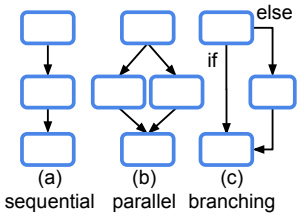

Figure 1: Simple DAGs that can be used to compose complex video pipelines.

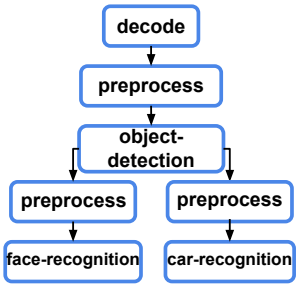

Figure 2: An AMBER Alert pipeline that finds faces and cars in a video.

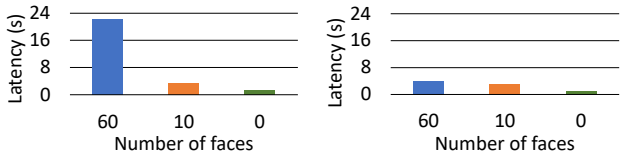

Figure 3: Execution latency on CPU (left) and GPU (right) for a face detection pipeline that identifies unique faces in a frame [50]. Latency varies up to 17.2× and 4× on CPU and GPU, respectively.

2 BACKGROUND AND MOTIVATION

Applications define *video pipelines* as directed acyclic graphs (DAGs), where vertices represent video analytics and processing operations, while edges represent dataflow.

As described in literature [4, 32, 48], video pipelines can be composed from three basic DAG patterns shown in Figure 1: (a) sequential, where each vertex has at most one input and one output, (b) parallel, where multiple vertices execute in parallel, and (c) branching, where the output of a vertex, called branching vertex, conditionally determines the next vertex to execute. For example, the AMBER Alert pipeline [63, 78] for face and car recognition in Figure 2 begins with a sequential path of decoding and preprocessing operations, followed by a branching object detection operation. Depending on the output, people or cars are sent to parallel face and car recognition operations, respectively.

Table 1 categorizes video analytics and processing systems based on two key features: (a) Their ability to specify and meet *performance targets*. User-facing systems typically require that the video pipeline meet a latency target, ideally while minimizing resource usage (cost). For example, the AMBER Alert pipeline needs to meet a strict latency target so that responders can take timely action. (b) Support for *general video operations*. To compose video pipelines, a user combines video operations (e.g., inference models, video encoders, and image filters) with analytics operations that process extracted metadata. For example, the AMBER Alert pipeline will contain video decoding, object detection, face recognition, and car model recognition. Some systems, such as Scanner [61], VideoStorm [78], and gg [35], support general video operations. Others, such as Focus [42], Nexus [63], GrandSLAm [48], and InferLine [28] focus on one facet of video pipelines (e.g., deep learning inference) and rely on external services for other operations.

2.1 Challenges

Large configuration space. Pipeline operations offer a variety of *knobs* that can be tuned to improve latency and resource use. For example, many operations have knobs such as batch size, sampling rate, and resolution. Other knobs select the hardware platform (e.g., CPU, GPU, TPU, etc.) and set the resource allocation (e.g., CPU cores or GPU memory). Determining configurations is challenging due to the exponential growth in the configuration space with the number of operations, knobs, and hardware platforms available.

As shown in Table 1, Scanner, gg, and Sprocket do not auto-tune configurations knobs, putting the burden on the user to statically specify good operation configurations. Focus, Nexus, GrandSLAm, and InferLine are domain-specific to deep learning inference and are limited to configuring the inference models used and the batch size. VideoStorm supports general knob configurations; however, it takes tens of CPU hours to profile pipelines and requires re-profiling when the pipeline, input video, or latency targets change [78].

Input-dependent execution flow. Input-dependent execution flow occurs in two cases: First, inputs determine the conditional path in branching pipelines. In the AMBER Alert pipeline of Figure 2, a frame will only take the face recognition path if `object-detection` finds a person in it. Since a conditional path is not resolved until the branching

operation finishes, provisioning resources and selecting configurations to meet a pipeline's latency target is challenging. Existing systems either treat branching pipelines as parallel ones (i.e., by executing all conditional branches) [28, 33, 61] or do not support non-sequential pipelines [22, 48, 78].

Second, operations can produce a variable number of outputs, and thus a variable load for downstream operations. If the number of intermediate outputs is unknown until the operation is executed, determining the parallelism or resources needed downstream to meet latency targets is difficult, especially if these operations are computationally expensive. Figure 3 shows the latency for a pipeline that identifies the unique faces in a frame depends on the number of unique faces: 17.2× and 4× difference between 60 faces versus no faces on a CPU and a GPU, respectively. Thus, the nondeterminism introduced by input-dependent behavior requires systems to dynamically adapt to meet a pipeline's latency target [33]. Most existing video analytics and processing systems do not account for input-dependent execution flow.

Dynamically adjusting resource allocation of operation invocations. As a pipeline executes, the degree of available parallelism depends on the various knob settings (e.g., batching) and the number of intermediate outputs. Many existing systems require users to statically provision a cluster, which limits the resources available to exploit parallelism [33, 48] or leads to over-allocation and higher costs when parallelism is low. Some systems periodically adjust resources and bin pack requests as the load changes, but are limited by how quickly hardware (e.g., GPUs) and VMs can be loaded/unloaded [63]. Systems like gg [35] and Sprocket [22] leverage serverless platforms [3, 5, 9] to dynamically allocate resources for each operation invocation. However, serverless platforms still require users to manually select hardware types and configure knobs to meet latency targets.

3 LLAMA DESIGN

Llama is a heterogeneous and serverless framework for autotuning video analytics and processing pipelines. Llama's objective is to meet the overall pipeline latency target, while minimizing cost (resource usage). As noted in Section 2, input-dependent execution flow and resource volatility preclude the use of static tuning approaches [33]. They also preclude designing and calculating a globally-optimal solution a-priori or dynamically. Instead, Llama optimizes the overall pipeline execution by iteratively and dynamically optimizing each operation invocation using the most up-to-date information about the state of execution flow and resource availability. Specifically, Llama (a) dynamically reduces the pipeline target latency to per-operation invocation latency targets, values that we call *slack*, and (b) continuously configures each operation invocation to meet the slack at minimal cost. Dynamically allotting slack ensures the pipeline latency target is met without having to statically account for all possible conditional paths or sources of resource volatility in serverless environments. It also allows Llama to revisit configuration decisions as the resource environment evolves or as input-dependent operations are run. Llama finds the set of cost-efficient configurations for the entire pipeline because it minimizes cost at each configuration assignment subject to the overall latency target.

We address the challenges outlined in Section 2 as follows:

Traversing the large configuration space. Llama profiles and makes configuration decisions on a *per-operation*, not per-pipeline basis. New operations undergo a short (seconds to minutes), one-time profiling step independent of the pipelines that include the operation. Operations are not re-profiled as the pipeline composition, video, or latency targets change. As the pipeline executes, Llama makes configuration decisions for one operation invocation at a time, reducing the exponential configuration space of an entire pipeline to that of an individual operation.

Handling input-dependent execution flow. Llama uses three techniques to meet latency targets despite the nondeterminism that stems from input-dependent behavior and resource volatility (e.g., resource contention): (a) *early speculation and late commit* selects an initial configuration decision as soon as an invocation is available, then revisits the configuration right before execution, (b) *priority-based commit* prioritizes operations based on their affinity to hardware and their depth in the pipeline, and (c) *safe delayed batching* waits for additional inputs for batching, as long as doing so does not violate the invocation's allotted slack.

Dynamically adjusting resource allocations. Making per-invocation configuration decisions also allows Llama to dynamically right-size resource allocations across heterogeneous serverless backends. Llama decides the hardware type and resource sizing (e.g., GPU with 2GB of memory) during dynamic configuration based on what is necessary to meet the slack. Early speculation and late commit, as well as priority-based commit, also allow Llama to balance resources between operations.

3.1 Architecture

Llama uses an offline specification phase and an online optimization phase (Figure 4). The specification phase has two purposes. First, it allows the user to specify a pipeline with multiple, general operations using an SDK. Second, it extracts the following information: a set of all possible sequential paths through the pipeline, and the latency and resource footprint of each unique operation across possible knob configurations. The pipeline specification and the extracted metadata are stored for use during the online phase.

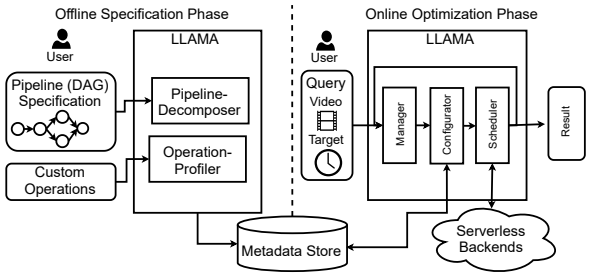

Figure 4: LLAMA's architecture diagram.

```
"engine": "gpu",
"resource": "200MB",
"latency": "320ms",
"cost": "$3.413e-6",
"arguments": [ "image_blur", "<input0>", "<input1>",
               "<output0>", "<output1> ],
"binary_name": "image_blur",
"batch_size": 2,
"num_inputs": 2,
"num_outputs": 2,
"resolution": 1920x1080,
"id": "image_blur_7"
```

Figure 5: Example configuration specification for one image blurring configuration. LLAMA's configuration specifications allow for general operation knob configuration.

The online phase is triggered when users submit an input video and a latency target to LLAMA. LLAMA executes the pipeline by continuously generating and executing a set of invocations for each operation as their input dependencies are resolved. For example, if `object-detection` outputs a frame tagged with a person, a new invocation is generated for the `preprocess` operation in the AMBER Alert pipeline (Figure 2). The online phase configures each invocation by first estimating its slack. It then uses the respective operation's profiling data to determine the most efficient configuration for completing the invocation within the allotted slack. The process repeats until all pipeline invocations are executed.

3.2 Specification phase

Application Interface. Users specify pipeline operations, dependencies between operations, and conditional flow using the LLAMA SDK. LLAMA provides a library of operations (e.g., decode and face recognition). Each operation consists of a binary executable, indexed by its SHA256 hash, and a *configuration specification* file that contains configuration options and performance statistics for the operation. Users can optionally bring their own operations by providing an executable and a configuration template that specifies tunable knobs (e.g., hardware type, batch size, or number of filters), the ranges for each knob, and the granularity of this range (e.g., batch size increasing by powers of 2). The Operation-Profiler uses these inputs in a one-time profiling step to generate a configuration specification. The operation and configuration specification are then added into the Metadata Store and re-used across pipelines without further profiling.

Operation-Profiler. The Operation-Profiler collects performance and resource statistics for each operation. Using the operation executable and configuration template as inputs, it first enumerates all possible configurations specified by the template, then executes a short profiling step using one or more sample frames for each configuration (depending on the batch size). Statistics such as latency and resource footprint (e.g., peak memory utilization) are collected and stored as configuration specification file entries. The frame content does not affect these statistics (recall that input-dependent execution flow manifests between operations). Since slack calculation (Section 4.1) is only dependent on the *relative* performance of operation invocations across the pipeline, the Operation-Profiler designates a *reference configuration* for each operation to provide a measure of relative performance. We chose the smallest CPU configuration (1-core, batch-1) for each operation's reference configuration. During runtime, operation invocation performance that differs from its profiled value, due to resource contention or profiling inaccuracy, is managed by leveraging feedback (Section 3.3).

An example configuration specification for an image blurring operation is shown in Figure 5. As shown, the configuration specification is structured so arbitrary operation- and hardware-specific configuration knobs can be described by users, and dynamically configured during runtime. This enables LLAMA to support general operations and arbitrary video pipelines for a myriad of applications.

Pipeline-Decomposer. To enable the online phase to dynamically compute slack, the Pipeline-Decomposer performs a one-time *decomposition* of the pipeline into all possible sequential paths in the pipeline. To do so, it performs a modified depth-first search on the pipeline DAG to enumerate all paths from the input operation (i.e., operation with no upstream dependencies) to an output operation (i.e., an operation with no downstream dependencies). It then emits an intermediate representation of the decomposed paths into the Metadata Store. For example, the AMBER Alert pipeline in Figure 2 is decomposed into the two sequential paths ending in `face-recognition` and `car-recognition`, respectively.

3.3 Online phase

Manager. LLAMA's Manager takes video inputs and latency targets and orchestrates the entire pipeline execution, maintaining execution state and generating new invocations. Whenever an invocation completes, the Manager records the invocation's runtime statistics (i.e., latency, cost, and configuration) and the location of intermediate outputs. The

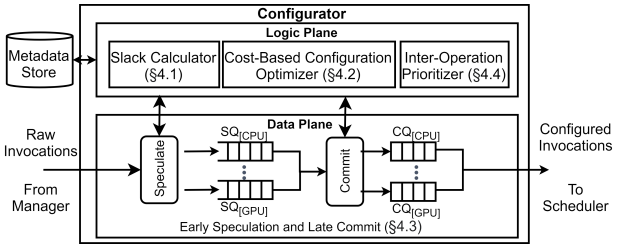

Figure 6: Configurator diagram.

runtime statistics are used to update the configuration profiles obtained from the Operation-Profiler via a feedback loop. We use an exponential smoothing algorithm to update the profiling; other algorithms can be incorporated as well. The intermediate outputs are then used to resolve any conditional branches. The Manager then spawns invocations for downstream operations once all dependencies have been resolved. These invocations are then sent to the Configurator.

Configurator. To meet the overall pipeline latency target, the Configurator (Figure 6) decides (a) how much slack to allot to an operation invocation, and (b) what the most efficient configuration is to meet the slack. The Configurator works with the Scheduler to keep track of available resources at a serverless backend as it makes configuration decisions.

Scheduler. After the Configurator has configured an invocation's knobs, the invocation is sent to the Scheduler for execution. The Scheduler executes the configured invocations on the hardware platform specified by the Configurator. This includes creating and managing the necessary backend connections, mitigating stragglers, and handling invocation failures (Section 4.5). When an invocation successfully returns, the Scheduler provides the Manager with the invocation metadata and output results.

4 TARGET LATENCY-AWARE CONFIGURATION

Input-dependent execution flow and backend resource volatility require the Configurator to dynamically determine each operation invocation's most efficient knob configurations. The Configurator is divided into two parts. The Logic Plane (a) determines how much slack can be spent on its invocation, and (b) uses a cost-based optimizer to select a configuration to meet that slack. The Data Plane manages configured operations in queues prior to their execution. Figure 6 presents how each of the key techniques discussed in this section is integrated into the Configurator.

Algorithm 1 Operation invocation slack allotment

1: $paths \leftarrow$ A set of all sequential paths in the pipeline
2: $t \leftarrow$ elapsed time
3: $target \leftarrow$ pipeline latency target
4: **procedure** ComputeSlack(op, λ)
5: $slacks = \{\}$
6: **for all** $\{path \in paths \mid op \in path\}$ **do**
7: $pLat =$ RemainingPathLatency$(op, path)$
8: $remainingTime = (target - t - queueingTime(\lambda))$
9: $pSlack = (op.ReferenceLat()/pLat) \times remainingTime$
10: $slacks.append(pSlack)$
11: **return** $min(slack \in slacks)$

4.1 Determining an invocation's slack

Given a user-specified pipeline latency target, the Configurator first needs to compute a slack for each operation invocation. Existing systems (e.g., GrandSLAm [33] and Fifer [40]) statically determine each operation invocation's slack by assuming a linear pipeline with predictable invocations and latencies (i.e., no nondeterminism). Our insight is to instead *dynamically* calculate each operation invocation's slack, which we subsequently use to select the best invocation configuration (Section 4.2). Doing so across invocations efficiently meets the pipeline latency target.

Llama calculates an operation invocation's slack using Algorithm 1. Given an invocation of operation *op* and a configuration's backend λ (e.g., a GPU cluster or a commercial serverless offering), ComputeSlack begins by finding every sequential path through the DAG containing *op* (Line 6). It then estimates the latency to complete the path, starting at *op*, using the reference configuration for each operation (Section 3.2). By using the reference configuration's latency, Llama avoids a causal dilemma of needing a configuration to compute slack, and needing a slack to select a configuration. The operation invocation's slack for that path is then determined based on the remaining time (Line 8), factoring in estimated queueing time at λ, weighted by the relative latency of *op* to the remaining path (Line 9). The final slack for an invocation of *op* on λ is then the minimum slack value over all possible execution paths of *op*, which accounts for all input-dependent branch resolutions (Line 11). We discuss how Llama reclaims overly-conservative slack next.

4.2 Navigating the configuration space

Since slack is calculated for each operation invocation, Llama can quickly evaluate configurations in a smaller per-operation, not per-pipeline, configuration space. After calculating the invocation's slack for each available serverless backend λ (Algorithm 1), Llama applies the objective function shown in Equation 1 for all possible configurations x of *op* using the invocation slack corresponding to the serverless

hardware backend $\lambda(x)$ targeted by x. $R(x)$ is the set of resources requested by x, and $R_{total}(\lambda(x))$ is the resource limit of $\lambda(x)$. $L(x)$ is the estimated latency to run configuration x. $C(x)$ is the estimated cost to run configuration x, and is computed as a weighted product of $R(x)$ and $L(x)$ Figure 5 shows an example of x for a GPU cluster, in which $R(x)$ is *resource* (200MB), $L(x)$ is *latency* (320ms), and $C(x)$ is *cost* ($3.413e-6). Here, $R_{total}(\lambda(x))$ would be the total aggregate GPU memory available in the cluster. $B(x)$ is the batch size of configuration x, and α is a tunable weight.

$$obj(x, slack) = \begin{cases} C(x)/B(x) & L(x) < slack \\ \frac{C(x)}{B(x)} + \alpha \frac{(L(x)*R(x))}{(B(x)*R_{total}(\lambda(x))} & otherwise \end{cases} \quad (1)$$

Intuitively, this objective function evaluates the monetary cost to run x when there is a feasible slack. If slack cannot be met (e.g., if the user submits an unachievably low target), the cost function weighs in favor of potentially more expensive configurations that achieve a higher throughput. α sets the balance between cost and throughput, with high values of α set to meet the slack at all costs, while lower values of α may leverage more cost-efficient configurations potentially at the expense of exceeding slack. The configuration objective function is independent of the input video or overall pipeline.

Users who wish to optimize for a different metric (e.g., minimal latency subject to a cost budget) can add their own objective function to Llama. Furthermore, since R is specific to each backend (e.g., concurrent invocation limits, GPU memory, or CPU cores), Llama supports heterogeneous backends (e.g., serverless GPUs or on-premise clusters). Llama also supports $R(x)$ being multiple resources. In this case, x is a vector (e.g., CPU cores and memory). To optimize for these resources, Llama can (a) combine the resources into one normalized scalar unit (similar to Amazon's EC2 Compute Unit [1]), (b) use multi-objective optimization to jointly evaluate the resources, or (c) separately evaluate each resource with the objective function and aggregate (e.g., sum) the objective function outputs.

Since conditional flow will not always resolve to the worst-case path, the allotted slack may result in a configuration with a lower-than-necessary latency. However, since each invocation is configured separately and dynamically, future invocations will recover efficiency from earlier mis-predictions.

4.3 Revisiting configuration decisions

To manage invocations that cannot be run concurrently due to limited backend parallelism, Llama locally queues invocations and accounts for the queueing time when allotting slack (Line 9 in Algorithm 1). The queueing time depends on x_i: the selected configuration of each queued invocation i (i.e., it is not sufficient to use the number of queued operation invocations as a measure of wait time [33, 66]). Thus, invocations need to be assigned a configuration before they are queued. However, the initial configuration x_i is often made many seconds before it is actually invoked, leading to sub-optimal configurations for three reasons. (a) Invocations queued *in front of* i may experience execution times that vary from the profiled values. This can occur due to resource contention or input-dependent execution flow. (b) The estimated latency for x_i may be updated via feedback while it is queued. (c) The number of invocations queued *behind* i may quickly grow (e.g., many completed `object-detection` invocations may output a large number of `car-recognition` and `face-recognition` invocations); thus, x_i should be chosen to ensure upstream invocations can meet their slack. Hence, by the time a queued invocation is ready to run, its selected configuration needs to be revisited to determine if it is still the right configuration.

To solve this, Llama leverages a novel technique inspired by late binding [25, 52, 53, 60, 67] that we call **early speculation and late commit**. With early speculation and late commit, Llama maintains two queues per serverless backend λ: an unbounded speculative queue ($SQ[\lambda]$) and a small, bounded commit queue ($CQ[\lambda]$) set to hold enough invocations to saturate λ. Once an invocation i is ready to execute, the Configurator uses Algorithm 1 to assign it a slack and uses Equation 1 to select a *speculative configuration*. The configured invocation is then put into the appropriate speculative queue, thus enabling Llama to estimate the queueing time at each backend. Once i reaches the head of the speculative queue, as prior invocations are executed, Llama revisits the configuration of i by using Algorithm 1 and Equation 1 again. It then *commits* the configuration into the appropriate commit queue. Doing so mitigates the queueing challenges we noted above by delaying binding an invocation to a final configuration for as long as possible. This provides Llama with maximum flexibility and the most up-to-date state about pipeline dataflow and performance at each backend.

With early speculation and late commit, Llama can estimate the queueing time using Equation 2 for each serverless backend λ based on each configured invocation i in its queues. $L(x_i)$ and $R(x_i)$ are the estimated latency of, and resources requested by x_i, respectively. $R_{total}(\lambda(x_i))$ is the total amount of resources or concurrency limit at the serverless backend specified by the configuration x_i.

$$Q_{SQ[\lambda],CQ[\lambda]} = \sum_{i \in \{SQ[\lambda],CQ[\lambda]\}} L(x_i) \frac{R(x_i)}{R_{total}(\lambda(x_i))} \quad (2)$$

Intuitively, the queueing time is the sum of each x_i's profiled configuration latency, weighted by x_i's requested resources (to account for parallel execution). The cumulative queueing time over $SQ[\lambda]$ and $CQ[\lambda]$ is then used in ComputeSlack. $SQ[\lambda]$ is included when committing configured

invocations to account for invocations queued behind i. We do not incorporate future operations' queueing time, since dynamicity and the need to assign a configuration to each downstream operation can result in inaccurate estimates.

4.4 Inter-operation prioritization

4.4.1 Challenges.

The Configurator's decisions described in Section 4.2 assume per-operation invocation decisions can be made independently of each other. However, Llama also needs to reason about the relationship between operations and their invocations for the following reasons:

When to batch invocations. As pipeline dataflow progresses, there can be moments when an operation may have fewer invocations available than the most efficient configuration's batch size. For example, if a pipeline contains a slower face detection operation followed by a faster blurring operation, the blur operation's invocations will likely drain the speculative queue faster than it can build up. In such cases, executing upstream operations first yields a larger batch size, amortizing RPC and I/O overheads. However, waiting for upstream operation invocations to complete their execution may result in a slack violation.

Under-allotting slack due to incorrect profiling. As described in Section 4.1, slack allotted to an invocation is a function of the reference configuration's profiled latency for downstream operations. Furthermore, a configuration's latency is updated using a feedback loop after execution (Section 3.3). However, slack can be under-allotted to operation invocations if the reference configuration latency is significantly shorter than its actual latency, and the feedback loop is not closed early on during pipeline execution. This is especially problematic for longer pipelines, and for pipelines with the last operation's invocations needing a longer slack than the rest. Thus, it is beneficial to prioritize invocations by pipeline depth early in the pipeline's execution so that feedback can update all reference configurations.

Affinity of operations to heterogeneous hardware. While prioritizing invocations by pipeline depth can help prevent under-allotting slack, the issue of prioritizing operation invocations on particular hardware platforms remains. For example, consider the case in which both an `object-detection` and `face-recognition` invocation must be configured. Assume that while both operations run faster on a GPU, `face-recognition` benefits more from acceleration and observes a larger latency reduction. Resource limits force the two invocations to split their decision between λ_{CPU} and λ_{GPU}. Committing `object-detection`'s invocation first forces `face-recognition`'s invocation to choose λ_{CPU}. However, the better decision is to assign the CPU to `object-detection` and the GPU to `face-recognition`. The relative benefit of running operation invocations on a particular hardware platform (i.e., its hardware affinity) must be incorporated into configuration decisions.

4.4.2 Our solution.

Llama addresses these challenges using **safe delayed batching** and **priority-based commit**, implemented in conjunction with early speculation and late commit.

Safe delayed batching. Safe delayed batching addresses the challenge of waiting for additional invocations to batch. During both early speculation and late commit, if Llama determines the most cost-efficient configuration that meets slack has a batch size larger than the number of invocations available for a given operation (using Equation 1), it waits until more invocations arrive to assign a configuration. It does so *safely* — only if there are enough upstream invocations and slack will not be violated. Otherwise, it uses the best feasible configuration.

Priority-based commit. Priority-based commit addresses the challenges of under-allotting slack and operations' affinity to heterogeneous hardware. First, to address the challenge of under-allotting slack, the Configurator prioritizes invoking a certain number of reference invocations for each operation, favoring deeper operations in the pipeline. This ensures the feedback loop for all reference configurations is closed as fast as possible to minimize under-allotted slack.

Second, to compute an operation's affinity to heterogeneous hardware, Llama compares the benefits an operation invocation receives from running on a specific backend to other available backends. It computes the *affinity* of an invocation's operation op to hardware backend λ using Equation 3, where $X_{op,\lambda}$ is the subset of configurations for op that run on λ and $X^{c}_{op,\lambda}$ is the complementary set (i.e., all other configurations for op).

$$affinity(op, \lambda) = \frac{\min_{\forall x \in X^{c}_{op,\lambda}} \{obj(x, slack)\}}{\min_{\forall x \in X_{op,\lambda}} \{obj(x, slack)\}} \quad (3)$$

Intuitively, Equation 3 determines if a hardware backend provides more benefit (via Equation 1) to an invocation than other backends. Llama prioritizes invocations from operations with a higher affinity to a hardware backend λ when committing them to each $CQ[\lambda]$. This ensures each backend achieves its highest utility.

4.5 Handling stragglers and invocation failures

During execution, operation invocations may straggle or fail to execute [21, 30, 74]. The Scheduler keeps track of each invocation's execution time. If it exceeds a configurable timeout (discussed in Section 5) or the Scheduler receives an error,

the Scheduler notifies the Manager to create a duplicate invocation. This duplicated invocation is then passed to the Configurator to begin the slack allotment and configuration process anew. The allotted slack will now be reduced, potentially resulting in a different configuration to still meet the pipeline latency target (evaluated in Section 6.4).

5 IMPLEMENTATION

We implemented LLAMA as an extension to gg in ~4K lines of C++ code. We modified gg's C++ and Python SDK to support complex pipelines and general knob configurations. LLAMA supports operations from any framework or library; we implemented non-deep learning pipeline operations (e.g., blur and meanshift) using OpenCV [26] and FFmpeg [34], and deep learning pipeline operations with TensorFlow [16].

We implemented the online phase on top of gg's dispatcher and backend resource manager. The online phase is single-threaded but can scale out to multiple threads as needed. For straggler mitigation, we set each invocation's time-out value to 1.5× the invocation's profiled latency. Larger values wait too long to spawn a duplicate invocation, which may violate the pipeline latency target, while smaller values unnecessarily overload the speculation queues. For depth-first priority, we observed that 10 invocations of the reference configuration were sufficient to obtain enough feedback values to converge on a latency measurement. Smaller values do not collect enough feedback values to prevent under-allotted slack, while larger values unnecessarily prioritize invocations with configurations that may not be efficient.

For the offline specification phase, we implemented the Operation-Profiler as a client to the online phase that collects and stores the profiled metadata into configuration specifications. Configuration specifications are implemented as JSON files. The Metadata Store is implemented in an object store (e.g., Google Cloud Storage).

We deployed LLAMA with serverless CPUs and serverless GPUs as compute backends. For serverless CPUs, we provision and manage a cluster of CPUs similar to existing serverless offerings [71]. Each invocation requests a specific number of cores (up to 4). LLAMA also supports running on serverless computing services such as AWS Lambda [3] or Google Cloud Functions [9], where the invocation resources requested would be an amount of DRAM.

Since there exists no serverless GPU services or frameworks at the time of writing, we built our own implementation (~1K lines of C++ code) that we believe is representative of a future production offering [27]. Similar to CPU serverless offerings, an invocation requests an amount of GPU memory (in MB) per invocation. Our serverless GPU scheduler then allocates a proportional amount of GPU threads using NVIDIA MPS [10], allowing for multiple invocations to execute concurrently. Invocations are executed on a first-come, first-served basis. LLAMA is also compatible with GPUs that support concurrent job execution in hardware [11].

6 EVALUATION

We answer the following questions: (a) How does LLAMA compare to state-of-the-art systems (Scanner, Nexus, gg, and GrandSLAm)? (b) How effective is LLAMA in meeting diverse latency targets? (c) How does each technique employed by LLAMA, such as early speculation and late commit and safe delayed batching, contribute to its ability to meet the latency target? (d) What is the impact of profiling errors and failures on LLAMA's ability to meet latency targets? (e) What are the overheads of various decisions LLAMA makes?

Metrics. Unless otherwise noted, we use pipeline processing latency and cost as metrics for success (similar to [22, 28, 48]). For each experiment, we report the mean of three runs.

Experimental setup. We deployed LLAMA on Google Cloud Platform (GCP) [8]. The LLAMA runtime ran on a `n1-standard-8` instance (8 vCPUs, 30 GB of DRAM). We used the following setup unless otherwise noted. For the serverless CPU backends, we used 10 `n1-standard-64` (64 vCPUs, 240GB of DRAM). For the serverless GPU backends, we used 2 `custom-12-46080` (1 V100 GPU, 12 vCPUs, 45 GB of DRAM). All instances feature Intel Xeon Platinum E5-2620 CPUs operating at 2.20GHz, Ubuntu 16.04 with 5.3.0 kernel, and up to 32 Gbps networking speed. The backends are sized to match each other in cost: `custom-12-46080` and `n1-standard-64` VMs are effectively priced the same on GCP (a difference of 1% at the time of writing). (This price-equivalency is also true for equivalent instances on AWS.) This allowed us to use the same compute resources for both LLAMA and the baselines.

Baseline systems. We compared LLAMA with three sets of systems: (a) cluster systems (Scanner and Nexus), (b) serverless systems (gg), and (c) target-aware systems (GrandSLAm). Scanner is used by Facebook for processing 360° videos [13]. Nexus accelerates deep learning-based video analysis on GPUs. gg is a general purpose serverless framework. GrandSLAm estimates slack to meet pipeline latency targets for sequential, DNN-only pipelines. We evaluated two common Scanner setups: one in which a user only provisions a cluster with CPUs (`sc-cpu`), and one in which, similar to Nexus, a user runs all operations on a GPU (`sc-gpu`). For gg, we also compared against a version augmented with LLAMA's branching support (`gg-branch`). `sc-cpu`, `gg`, and `gg-branch` do not support heterogeneous accelerators, while `Nexus` and `sc-gpu` require GPU VMs. Since GrandSLAm does not natively support non-sequential pipelines, and does not account for input-dependent execution flow, we implement it with LLAMA by disabling early speculation and late commit,

Pipeline	Description	Length (Form)	Operations (# of total configurations)	Video input
AMBER Alert	detect cars and people	5 (branching)	decode†, preprocess†, object detect., face recog., car recog. (646)	traffic camera [15], 10 min, 1080p
Face Blurring	detect indiv. face and blur from all frames	5 (branching)	decode†, preprocess†, face recog., template match†, blur† (600)	rally [12], 10 min, 720p
Denoising	detect indiv. face and denoise/segment	5 (branching)	decode†, preprocess†, face recog., template match†, meanshift† (600)	rally [12], 10 min, 720p
Toonify	apply cartoon effect to video	4 (parallel)	decode†, edge detect.†, bilateral filter†, merge edge-filter†, encode† (989)	tears of steel [14], 10 min, 720p
Synthetic	synthetic pipeline for sensitivity analysis	7 (sequential)	decode†, blur†, preprocess†, face recog. (596)	rally [12], 10 min, 720p

Table 2: Video pipelines used for evaluating Llama, their operations, and video inputs. † are non-deep learning pipeline operations.

feedback, and depth-first priority. However, GrandSLAm still has access to Llama's branching support, safe delayed batching, priority-based commit, and dynamic resource allocation across heterogeneous backends (`GrandSLAm++`). To equalize the compute resources provided to all systems, we provisioned `sc-cpu`, `gg`, `gg-branch`, and `GrandSLAm++` with 12 `n1-standard-64`, and `Nexus` and `sc-gpu` with 12 `custom-12-46080`.

Resource requests and cost model. For Llama, gg, and `GrandSLAm++`, each invocation requests a set amount of resources (GPU memory or CPU cores) as is done in commercial serverless offerings. The respective backend then provisions the invocation with the requested resources, and charges a price based on the amount of requested resources and invocation latency. We calculate the price (in $/(resource-second)) by dividing the cost per second charged by GCP by the VM's total resources. For example, the price of a V100 GPU invocation is calculated by dividing the price of `custom-12-46080` by 16*GB*. Since Scanner and Nexus are cluster-based frameworks, we compute cost using the time to rent the cluster for the duration of the execution; we do not include the cost of starting and maintaining a warm cluster.

Pipelines, operations, and videos. Table 2 shows the pipelines, operations, and videos that we used. For branching pipelines, only invocations satisfying the branching condition are executed. For AMBER Alert, only frames with faces and cars execute their respective recognition paths. For Face Blurring and Denoising, frames with faces proceed to a template match operation where the frame is compared against a pre-determined face. If a match is found, the face in the frame is then blurred, or denoised using meanshift. The Toonify pipeline executes the bilateral filtering and edge operations in parallel before merging and encoding the frames. Finally, the synthetic pipeline is a chain of 5 image blurring operations followed by a face recognition operation. The face recognition operation is the most compute-intensive operation of this pipeline, which allows us to evaluate Llama's ability to meet diverse pipeline latency targets, even when configurations were mis-profiled (Section 6.4). Since `sc-cpu`, `sc-gpu`, and gg do not support branches, they execute the three branching pipelines as parallel ones (i.e., both branches are executed). Videos were selected based on their use in similar pipelines in prior work (e.g., tears of steel [14] in Sprocket [22]) or based on their content (e.g., a traffic video to exercise branches in AMBER Alert).

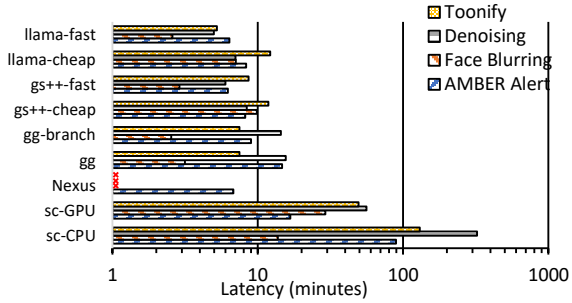

Figure 7: Latency of baselines to execute each pipeline. Nexus only supports the AMBER Alert pipeline (unsupported pipelines are denoted by ×). Llama's fastest execution is faster than all baselines.

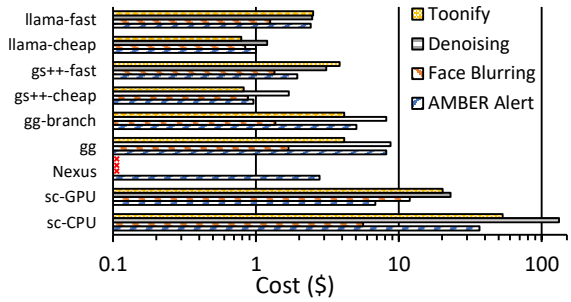

Figure 8: Cost incurred by baselines for each pipeline. Nexus only supports the AMBER Alert pipeline (unsupported pipelines denoted by ×). Llama's cheapest execution is cheaper than all baselines.

6.1 Comparing LLAMA to existing systems

We first show how LLAMA's ability to dynamically reconfigure operation invocations enables it to outperform existing systems, both in terms of latency and cost.

Experimental setup. For Nexus, we set the pipeline latency target to be 2 seconds per frame, which we found to be the strictest latency that does not drop any requests [63]. Nexus then automatically configures the batch size and number of instances for each model. For sc-cpu and sc-gpu, we swept each operation's batch size from 1 to 64 (by powers of 2) and set each value based on the lowest pipeline execution latency (reported in Figure 7). For gg and gg-branch, we set each invocation's configuration based on the lowest, most cost-effective CPU latency reported by the Operation-Profiler. We configured LLAMA and GrandSLAm++ with two pipeline latency targets: an unachievable low target (0 seconds) that forced both to minimize pipeline execution latency at the expense of cost: llama-fast and GrandSLAm++-fast, and an overly-loose target (∞ seconds) that allowed LLAMA to minimize the overall cost: llama-cheap and GrandSLAm++-cheap.

Results and discussion. Figures 7 and 8 show the processing latency and total cost, respectively, to execute each of the four non-synthetic pipelines. LLAMA achieves lower latency, higher throughput, and lower cost than existing systems.

Even when the cost of starting and maintaining a warm cluster are not considered, LLAMA is faster (up to 65× and 28× on average) and cheaper (up to 110× and 55× on average) than sc-cpu. Compared to sc-gpu, LLAMA is up to 11× faster (6× on average) and up to 27× cheaper (18× on average). Scanner cannot dynamically adjust and right-size invocation configurations, and thus cannot address performance degradation caused by resource contention for compute-intensive operations (e.g., deep learning inference and meanshift) or memory-intensive operations (e.g., bilateral filtering).

Next, since Nexus focuses on inference-serving pipelines, we are only able to run the AMBER Alert pipeline (other pipelines denoted by × in Figures 7 and 8). While we provide Nexus with 12 GPUs, Nexus's bin-packing algorithm [63] utilizes only 8; thus, we report cost for 8 GPUs. By dynamically choosing CPU versus GPU configurations, LLAMA achieves 1.3× speedup and 2.8× lower cost compared to Nexus.

Compared to gg, LLAMA is up to 3.1× faster (2.2× on average) and up to 8.2× cheaper (5.7× on average). Compared to gg-branch, LLAMA is up to 2.9× faster (1.8× on average) and up to 6.8× cheaper (4.7× on average). While gg-branch can reason about conditional flow, it cannot make dynamic invocation configuration decisions or adjust to resource volatility, resulting in a higher latency and cost compared to LLAMA.

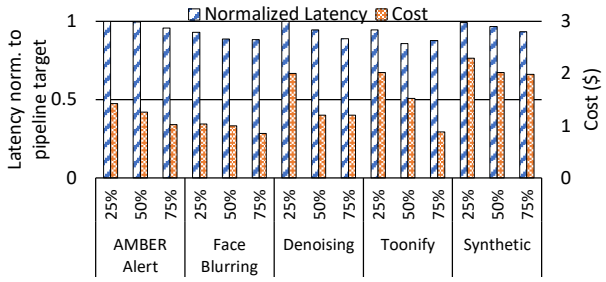

Figure 9: Evaluating LLAMA given varied latency targets. 50%: mean of the measured latencies of llama-fast and llama-cheap, 25%: mean of llama-fast and 50%, and 75%: mean of llama-cheap and 50%. The execution latency is normalized to the pipeline target (≤1 means target was met). Cost is in dollars. LLAMA meets all latency targets and reduces overall cost for less stringent targets.

Finally, LLAMA is up to 1.7× faster (1.2× on average) than GrandSLAm++-fast and 1.4× cheaper (1.1× on average) than GrandSLAm++-cheap. For the AMBER Alert pipeline, LLAMA's initial exploration using the depth-first priority technique led to a higher cost, but similar latency, as GrandSLAm++-fast, since LLAMA converged on a similar configuration. Since GrandSLAm++ allots slack and selects configurations based on profiled values, it cannot dynamically adjust to nondeterminism, which can result in slower performance or higher cost (e.g., Denoising).

By making dynamic invocation configurations, LLAMA can determine how well operations perform across heterogeneous backends and right-size resources depending on the pipeline latency target.

General applicability. While LLAMA was designed to address the challenges of running video analytics and processing pipelines (Section 2), its operation configuration specification (Section 3.2) supports arbitrary operation- and hardware-specific configuration knobs. To demonstrate this, we built a four-stage natural language processing pipeline for applications like therapy session analysis for at-risk youth [38]. The pipeline has six models (language identification, two language translation, sentiment analysis, text generation, and summarization) and features branching, parallel, and sequential patterns. For a 256-line transcript, llama-fast takes 275s ($1.22) while llama-cheap takes 573s ($0.44). Thus, LLAMA can be used for meeting latency targets for general domains.

6.2 Can LLAMA trade off latency for cost?

We now show LLAMA can also meet latency targets that lie between llama-fast and llama-cheap.

Experimental setup. For each pipeline, we provide three latency targets to LLAMA that lie between the times required to execute the pipeline using llama-fast and llama-cheap. The 50% latency target is the mean latency between the

Pipeline	# configs. used	% invoc. that met slack
AMBER Alert	27 ± 8	92% ± 6%
Face Blurring	29 ± 3	93% ± 1%
Denoising	40 ± 4	99% ± 0%
Toonify	30 ± 5	97% ± 3%
Synthetic	50 ± 16	88% ± 3%

Table 3: Mean and standard deviation of number of configurations used and percent of invocations that met their allotted slack. Llama accurately allots and meets almost all slack by selecting a variety of different configurations per pipeline.

latencies achieved by `llama-fast` and `llama-cheap`. The 25% latency target (the most stringent of the three) is the mean latency between `llama-fast` and the 50% latency target. Finally, the 75% latency target (the least stringent of the three) is the mean latency between `llama-cheap` and the 50% latency target. For example, `llama-fast` executed Face Blurring in 155 seconds, and `llama-cheap` executed it in 423 seconds; the 25%, 50%, and 75% latency targets are 225, 290, and 380 seconds respectively.

Results and discussion. Figure 9 shows the observed execution latency, normalized to each of the aforementioned pipeline latency targets (≤1 means that the latency target was met), as well as the raw cost values for each pipeline execution. Llama not only meets all latency targets, but also dynamically adjusts its configuration decisions to choose cost-efficient configurations as the latency target became less stringent. For the Denoising and Synthetic pipelines, the cost stays the same for the 50% and 75% targets. This is due to Llama selecting similar invocation configurations during both runs, since it determined them to be the most cost-efficient configurations for both latency targets.

Table 3 shows a breakdown of how many configurations were used to meet the 50% pipeline latency target, and what percent of invocations met the slack. We note that (a) Llama meets the slack for 94% of invocations on average across all pipelines, with the lowest being the Synthetic pipeline since it is the longest, and (b) the number of configurations used varies per pipeline. Thus, Llama's slack allotment and configuration selection algorithms (Section 4) are effective in meeting pipeline latency targets while minimizing cost.

6.3 Ablation study of Llama's techniques

We now show how each technique of Llama contributes to its ability to efficiently meet pipeline latency targets.

Experimental setup. We performed an ablation study with two distinct pipelines: Amber Alert and Toonify. Following is the list of techniques employed by Llama: feedback loop (FB, Section 3.3), depth-first priority (DFP, Section 4.4), safe delayed batching (SDB, Section 4.4), early speculation and late commit (ESLC, Section 4.3), and priority-based commit (PBC, Section 4.4). Note that priority-based commit includes

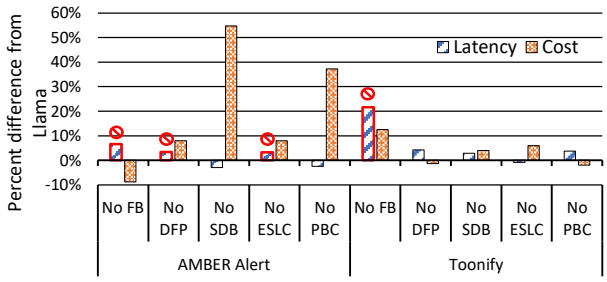

Figure 10: Impact of turning Llama's techniques off on the AMBER Alert and Toonify pipelines. Red borders and circled slashes indicate the pipeline latency target was violated. FB is feedback, DFP is depth-first priority, SDB is safe-delayed batching, ESLC is early speculation and late commit, and PBC is priority-based commit.

both depth-first priority and hardware affinity. For each run, we turn off a single technique and record the pipeline execution latency and cost. For each pipeline, we use its 50% pipeline latency target specified in Section 6.2.

Results and discussion. Figure 10 shows the results of our ablation study (red borders and circled slashes indicate the latency target was violated). For the AMBER Alert pipeline, disabling feedback, depth-first priority, or early speculation and late commit results in latency target violations. All three techniques allow Llama to accurately measure and adapt to performance volatility caused by input-dependent execution flow (branching operations) and resource contention. For example, disabling feedback causes Llama to miss the latency target because resource contention resulted in invocations taking longer than their profiled values. With feedback enabled, Llama is able to detect this and choose configurations with higher throughput at a small expense of cost-efficiency. On the other hand, disabling safe delayed batching or priority-based commit causes Llama to not use large batches for deep learning inference invocations on GPUs, resulting in reduced cost-efficiency.

For the Toonify pipeline, disabling feedback also causes a latency target violation similar to the AMBER Alert pipeline. Disabling either safe delayed batching or early speculation and late commit results in Llama choosing less cost-efficient configurations. On the other hand, disabling depth-first priority and priority-based commit results in more cost-efficient configurations without violating the latency target. This is because these techniques led to Llama choosing configurations that are more throughput-intensive than necessary for merge edge-filter operation invocations in an effort to meet the pipeline latency target. However, as noted for the AMBER Alert pipeline and evaluated in Section 6.4, both depth-first priority and priority-based commit are important for Llama's robustness in right-sizing resources and meeting latency targets despite profiling errors.

Pipeline (target)	Llama	Llama w/o FB & DFP
Denoising (350s)	(348s, $1.20)	(369s, $1.64)
Synthetic (520s)	(520s, $2.31)	(487s, $3.14)

Table 4: Impact of profiling errors. Latency and cost for the Denoising and Synthetic pipelines when profiled values are inaccurate (set to 50% of their measured latencies). FB is feedback and DFP is depth-first priority. Without these techniques, Llama cannot meet the latency target, or uses configurations that are not cost-effective.

6.4 Meeting targets despite profiling errors & failures

We now show that Llama can meet targets despite profiling errors and invocation failures.

Experimental setup. To evaluate "mis-profiling", all operation profiled latencies are set to 50% of their values. Separately, to evaluate Llama's resiliency to failures, we forced 3% of invocations to fail (2, 114 and 3, 617 failures for the Denoising and Synthetic pipeline, respectively). For both experiments, we used the Denoising and Synthetic pipelines because they represent worst-case scenarios: an expensive operation at the end of the pipeline with an under-estimated latency. In addition, the Synthetic pipeline is the longest pipeline, which further exacerbates profiling errors: Llama will under-allot slack to the last operation unless techniques are used to mitigate mis-profiling. We use the respective 50% pipeline latency target from Section 6.2 for each pipeline.

Results and discussion. Table 4 shows the impact of profiling error on latency and cost with (a) all of Llama's techniques, and (b) both feedback and depth-first priority turned off (the two techniques Llama relies on to adjust for inaccurate profiling). For the Denoising pipeline, disabling feedback and depth-first priority causes Llama to under-allot slack to the last meanshift operation. This results in a missed pipeline latency target because Llama could not adjust to the profiling errors until late in the pipeline execution. For the Synthetic pipeline, when both techniques were off, Llama meets the latency target but at a 35% higher cost. This is because the 50% lower-than-profiled latencies cause Llama's objective function (Equation 1) to incorrectly calculate that the CPU, not the GPU, is most cost-efficient for the meanshift operation. Even though latency is reduced due to the availability of more CPU resources, each CPU configuration was less cost-effective than a GPU, resulting in an increased cost.

When evaluating invocation failures, both pipelines were able to meet the specified latency target despite the high failure rate using the techniques described in Section 4.5.

These results demonstrate depth-first priority and feedback are necessary to resolve profiling discrepancies early on during execution, and that Llama is robust to failures.

Phase	Action	Latency (% of exec.)
Specification	Profiling	257 ± 155 s
	Path decomposition	1.74 s
Online	Speculate	0.005 ± 0.005 ms (0.08%)
	Commit	0.186 ± 0.813 ms (3.1%)
	Invoke	0.151 ± 0.078 ms (2.5%)
	Finalize	0.141 ± 1.209 ms (2.4%)

Table 5: Llama's decision overheads. Mean and standard deviation latencies of invocations for the AMBER Alert pipeline. Latencies are per-invocation for online actions, per-operation for profiling, and per-pipeline for path decomposition. For each online action, we show the percent of the execution time spent on the action across all operation invocations (73K).

6.5 Overheads of decisions Llama makes

Finally, we evaluate Llama's overheads and its ability to scale across backends.

Table 5 shows the overhead for these decisions when specifying and running the AMBER Alert pipeline with the 50% intermediate latency target from Section 6.2; all other pipelines have similar overheads. For the specification phase, profiling each operation takes an average of 257 seconds, and only needs to be performed the first time an operation is added to the Metadata Store. The decomposition step, which is performed once per pipeline, takes only 1.7 seconds.

During the online phase, Llama only spends 483 micro-seconds, on average, to process (i.e., speculate, commit, invoke, and finalize) an invocation, allowing Llama to process over 2000 invocations per second. Calculating a slack and determining a configuration is efficient, as speculate only requires 5 micro-seconds. Most time is spent evaluating priority between operations during commit, connecting and sending invocations to backends during invoke, and updating global state once invocations completed during finalize.

Low overheads also allow Llama to improve execution latency as the number of resources or maximum concurrency increases. Compared to `llama-fast` for the AMBER Alert pipeline run on 10 CPU and 2 GPU instances (Section 6.1), having 6 CPU and 1 GPU instances is 46% slower, while having 15 CPU and 3 GPU instances is 25% faster.

7 RELATED WORK

Video and general-purpose analytics frameworks. In Section 2, we describe the limitations of several existing video analytics and processing frameworks [22, 28, 35, 42, 61, 63, 78]. Other cluster-based and serverless systems for both domain-specific and general-purpose applications [33, 43, 46, 55, 59, 73, 77] either do not support independent-dependent execution flow, require extensive per-pipeline profiling, or require users to configure and right-size resources. Dremel [56], Google's BigQuery backend, was one of the first framework

to provide users with a fully-managed experience over their data. Similar to Llama, Dremel adaptively scales resources based on the execution DAG, and leverages the idea that disaggregated compute and storage resources can be managed in a serverless manner for users. However, unlike Dremel, Llama must also configure operations from a large design space, must consider heterogeneous serverless backends, and must meets diverse pipeline latency targets across complex (and possibly dynamic) video pipelines.

Dataflow optimizations and scheduling techniques. GrandSLAm [48] and Fifer [40] use slack to statically determine the batch size for sequential microservice graphs. Delayed batching is used by Clipper [29] to increase efficiency of inference queries, but must be statically set by users. Late binding is used by schedulers [25, 52, 53, 60, 67] to maximize the flexibility of the scheduling decision and knowledge of system state. However, these systems do not consider the need to configure operations for meeting pipeline latency targets. TetriSched [66] uses a scheduler that prevents tasks from being sent to a sub-optimal set of resources due to resources being held by earlier jobs, but only supports per-operation targets, not an end-to-end pipeline latency target. Early speculation and late commit, and priority-based commit allow Llama to compute slack and make configuration decisions for arbitrarily complex pipelines to meet overall pipeline latency targets. Musketeer [39] and Dandelion [62] optimize dataflow DAGs for execution on a broad range of execution engines or hardware platforms. These optimizations are compatible with Llama, and can be used to expand the backends and hardware platforms Llama supports.

Existing work in domains such as compilers, adaptive query processing, and feature-based web serving have also proposed algorithms for meeting latency targets given a DAG. Certain compilers leverage an inspector/executor model [24, 57] to encode irregular accesses in the DAG and use an online executor to assign compute based on measured runtime statistics to optimize parallelism. In feature-based web serving, existing work [76] has proposed algorithms for not only selecting services from sequential DAGs, but also from DAGs with input-dependent execution. Similar to Llama, it ensures that all sequential paths through the DAG can meet the latency configuration (i.e., rather than choosing the path with the highest probability of execution). Adaptive query processing [31] techniques, such as dynamically partitioning queries to queues to estimate resource costs and later adapting configurations based on runtime state, are used in various query engines [43, 56, 58, 59]. However, these existing solutions differ from this work since Llama must (a) configure invocations from a large design space (e.g., DNN batch sizes in addition to provisioning resources), (b) consider heterogeneous hardware backends, and (c) account for non-determinism from input-dependent execution flow and resource volatility.

Cost-based query optimization. Several works have explored cost-based query optimization for relational databases [17, 41, 49, 64, 65, 69], including for queries whose optimal plan is input-dependent [72]. Llama is compatible with these frameworks, and can leverage their optimizations as an extension to how configurations are selected.

Auto-tuning configurations. CherryPick [18] and Ernest [68] present a performance prediction framework for recurring data analytics jobs; however, these systems require tens of executions of a job to set the configuration parameters. PARIS [75] focuses on VM-size selection; OptimusCloud [54] and Selecta [51] are domain-specific VM configuration systems for databases and storage technologies, respectively. Llama dynamically configures general video operations to meet diverse latency targets, and only requires one-time per-operation profiling.

8 CONCLUSION

We presented Llama, a heterogeneous and serverless video analytics and processing framework that executes general video pipelines, meeting user-specified performance targets at minimal cost. By dynamically configuring individual operation invocations, Llama efficiently traverses large configuration spaces, adapts to input-dependent execution flow, and dynamically allocates resources across heterogeneous serverless backends. Llama makes per-operation invocation decisions by first calculating invocation slack, then leveraging techniques such as safe delayed batching, priority-based commit, and early speculation and late commit to efficiently and accurately select configurations that meet the slack. Llama achieves an average improvement of 7.8× for latency and 16× for cost compared to state-of-the-art systems.

Acknowledgements

We thank our shepherd, Eugene Wu, and the anonymous reviewers for their helpful feedback. We also thank Daniel Kang and members of the MAST research group for their insightful discussions to improve this work. This work was supported by the Stanford Platform Lab and its industrial affiliates, and the SRC Jump program (CRISP center). Mark Zhao is supported by a Stanford Graduate Fellowship.

REFERENCES

[1] 2021. Amazon ECU. https://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it.
[2] 2021. Ambarella CVFlow Architecture. https://www.ambarella.com/technology/#cvflow.
[3] 2021. AWS Lambda. https://aws.amazon.com/lambda/.
[4] 2021. AWS Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html.

[5] 2021. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[6] 2021. Cisco Annual Internet Report (2018-2023). https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[7] 2021. CNN - Futuristic cop cars may identify suspects. https://money.cnn.com/2017/10/19/technology/future/police-ai-dashcam/index.html.

[8] 2021. Google Cloud. https://cloud.google.com/.

[9] 2021. Google Cloud Functions. https://cloud.google.com/functions.

[10] 2021. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[11] 2021. NVIDIA A100 GPU. https://www.nvidia.com/en-us/data-center/a100/.

[12] 2021. Political Rally Video. https://www.youtube.com/watch?v=FGDFAD3Jkuc.

[13] 2021. Scanner. http://scanner.run/.

[14] 2021. Tears of Steel. https://www.youtube.com/watch?v=tjgM6ckoz88.

[15] 2021. Traffic Footage. https://www.youtube.com/watch?v=MNn9qKG2UFI.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[17] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (June 2012), 968–979. https://doi.org/10.14778/2336664.2336670

[18] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[19] Amazon Go 2021. Amazon Go. https://www.amazon.com/b?ie=UTF8&node=16008589011.

[20] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha. 2017. Real-Time Video Analytics: The Killer App for Edge Computing. *Computer* 50, 10 (2017), 58–67. https://doi.org/10.1109/MC.2017.3641638

[21] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 185–198. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan

[22] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[23] Artificial Intelligence Security Surveillance Cameras 2018. Artificial Intelligence Security Surveillance Cameras. https://www.theverge.com/2018/1/23/16907238/artificial-intelligence-surveillance-cameras-security.

[24] Ayon Basumallik and Rudolf Eigenmann. 2006. Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, New York, USA) *(PPoPP '06)*. Association for Computing Machinery, New York, NY, USA, 119–128. https://doi.org/10.1145/1122971.1122990

[25] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 20, 15 pages. https://doi.org/10.1145/3190508.3190532

[26] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[27] Jack Choquette and Wishwesh Gandhi. 2020. NVIDIA's A100 GPU: Performance and Innovation for GPU Computing. In *2020 IEEE Hot Chips 32 Symposium (HCS), Virtual, August 16-18, 2020*. IEEE.

[28] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285

[29] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[30] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 10.

[31] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends Databases* 1, 1 (Jan. 2007), 1–140.

[32] T. Elgamal. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 300–312. https://doi.org/10.1109/SEC.2018.00029

[33] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/2168836.2168847

[34] FFmpeg 2021. FFmpeg. https://ffmpeg.org/.

[35] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 475–488.

[36] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 363–376.

[37] Ilya Ganusov and Mahesh Iyer. 2020. Agilex Generation of Intel FPGAs. In *2020 IEEE Hot Chips 32 Symposium (HCS), Virtual, August 16-18, 2020*. IEEE.

[38] James Gibson, David Atkins, Torrey Creed, Zac Imel, Panayiotis Georgiou, and Shrikanth Narayanan. 2019. Multi-label Multi-task Deep Learning for Behavioral Coding. *IEEE Transactions on Affective Computing* (2019), 1–1. https://doi.org/10.1109/TAFFC.2019.2952113

[39] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: All for One, One for All in Data Processing Systems. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 2, 16 pages. https://doi.org/10.1145/2741948.2741968

[40] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 280–295. https://doi.org/10.1145/3423211.3425683

[41] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, What-If Analysis, and Cost-Based Optimization of MapReduce Programs. *Proc. VLDB Endow.* 4, 11 (Aug. 2011), 1111–1122. https://doi.org/10.14778/3402707.3402746

[42] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 269–286. https://www.usenix.org/conference/osdi18/presentation/hsieh

[43] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. Association for Computing Machinery, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[44] Fei Jiang, Yong Jiang, Hui Zhi, Yi Dong, Hao Li, Sufeng Ma, Yilong Wang, Qiang Dong, Haipeng Shen, and Yongjun Wang. 2017. Artificial intelligence in healthcare: past, present and future. *Stroke and Vascular Neurology* 2, 4 (2017), 230–243. https://doi.org/10.1136/svn-2017-000101 arXiv:https://svn.bmj.com/content/2/4/230.full.pdf

[45] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: Scalable Adaptation of Video Analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 253–266. https://doi.org/10.1145/3230543.3230574

[46] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[47] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12. https://doi.org/10.1145/3079856.3080246

[48] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 34, 16 pages. https://doi.org/10.1145/3302424.3303958

[49] Sunghwan Kim, Taesung Lee, Seung-won Hwang, and Sameh Elnikety. 2018. List Intersection for Web Search: Algorithms, Cost Models, and Optimizations. *Proc. VLDB Endow.* 12, 1 (Sept. 2018), 1–13. https://doi.org/10.14778/3275536.3275537

[50] Davis E. King. 2009. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research* 10 (2009), 1755–1758.

[51] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 759–773. https://www.usenix.org/conference/atc18/presentation/klimovic-selecta

[52] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2020. Sol: Fast Distributed Computation Over Slow Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 273–288. https://www.usenix.org/conference/nsdi20/presentation/lai

[53] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. 2018. Dynamic Query Re-Planning Using QOOP. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 253–267.

[54] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 189–203. https://www.usenix.org/conference/atc20/presentation/mahgoub

[55] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[56] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* (2020).

[57] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. 1988. Principles of Runtime Support for Parallel Processors. In *Proceedings of the 2nd International Conference on Supercomputing* (St. Malo, France) *(ICS '88)*. Association for Computing Machinery, New York, NY, USA, 140–152. https://doi.org/10.1145/55364.55378

[58] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[59] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 113–126.

[60] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of*

the Twenty-Fourth ACM Symposium on Operating Systems Principles (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[61] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. 2018. Scanner: Efficient Video Analysis at Scale. *ACM Trans. Graph.* 37, 4, Article 138 (July 2018), 13 pages. https://doi.org/10.1145/3197517.3201394

[62] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 49–68. https://doi.org/10.1145/2517349.2522715

[63] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[64] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-Based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 307–319. https://doi.org/10.14778/3368289.3368296

[65] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. IBTune: Individualized Buffer Tuning for Large-Scale Cloud Databases. *Proc. VLDB Endow.* 12, 10 (June 2019), 1221–1234. https://doi.org/10.14778/3339490.3339503

[66] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems, EuroSys 2016 (Proceedings of the 11th European Conference on Computer Systems, EuroSys 2016)*. Association for Computing Machinery, Inc. https://doi.org/10.1145/2901318.2901355 11th European Conference on Computer Systems, EuroSys 2016 ; Conference date: 18-04-2016 Through 21-04-2016.

[67] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. 2014. The Power of Choice in Data-Aware Cluster Scheduling. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 301–316.

[68] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[69] Stratis D. Viglas and Jeffrey F. Naughton. 2002. Rate-Based Query Optimization for Streaming Information Sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/564691.564697

[70] Martin Voogel, Yohan Frans, and Matt Ouellette. 2020. Xilinx Versal Premium Series. In *2020 IEEE Hot Chips 32 Symposium (HCS), Virtual, August 16-18, 2020*. IEEE.

[71] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[72] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, and Jingren Zhou. 2020. Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 14–27. https://doi.org/10.14778/3421424.3421427

[73] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. 2018. VideoChef: Efficient Approximation for Streaming Video Processing Pipelines. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 43–56. https://www.usenix.org/conference/atc18/presentation/xu-ran

[74] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. 2014. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2671005

[75] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. 2017. Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, 452–465. https://doi.org/10.1145/3127479.3131614

[76] Tao Yu, Yue Zhang, and Kwei-Jay Lin. 2007. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. *ACM Trans. Web* (2007).

[77] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.

[78] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 377–392. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang