Privacy Budget Scheduling

Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, and Roxana Geambasu, *Columbia University;* Mathias Lécuyer, *Microsoft Research*

https://www.usenix.org/conference/osdi21/presentation/luo

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Privacy Budget Scheduling

Tao Luo[*]
Columbia University

Mingen Pan[*]
Columbia University

Pierre Tholoniat[*]
Columbia University

Asaf Cidon
Columbia University

Roxana Geambasu
Columbia University

Mathias Lécuyer
Microsoft Research

Abstract

Machine learning (ML) models trained on personal data have been shown to leak information about users. Differential privacy (DP) enables model training with a guaranteed bound on this leakage. Each new model trained with DP increases the bound on data leakage and can be seen as consuming part of a *global privacy budget* that should not be exceeded. This budget is a scarce resource that must be carefully managed to maximize the number of successfully trained models.

We describe *PrivateKube*, an extension to the popular Kubernetes datacenter orchestrator that adds privacy as a new type of resource to be managed alongside other traditional compute resources, such as CPU, GPU, and memory. The abstractions we design for the privacy resource mirror those defined by Kubernetes for traditional resources, but there are also major differences. For example, traditional compute resources are replenishable while privacy is not: a CPU can be regained after a model finishes execution while privacy budget cannot. This distinction forces a re-design of the scheduler. We present *DPF* (*Dominant Private Block Fairness*) – a variant of the popular Dominant Resource Fairness (DRF) algorithm – that is geared toward the non-replenishable privacy resource but enjoys similar theoretical properties as DRF.

We evaluate PrivateKube and DPF on microbenchmarks and an ML workload on Amazon Reviews data. Compared to existing baselines, DPF allows training more models under the same global privacy guarantee. This is especially true for DPF over Rényi DP, a highly composable form of DP.

1 Introduction

Increasing evidence suggests that machine learning (ML) models trained on sensitive, personal information – such as auto-complete models trained on users' emails – expose individual entries from their training sets [8, 57]. Despite the evidence, there is an increasing trend to push models to end-user devices for faster predictions [6, 27, 54], share them across teams in a company [36, 56] and even externally [2, 43].

Differential privacy (DP) [15] promises to enable safe sharing of models by providing solid guarantees regarding the exposure of individuals' data through these models. DP randomizes a computation over a dataset (e.g. training one model) to bound the leakage of individual entries in the dataset through the output of the computation (the model). Each new DP computation increases this bound over data leakage, and can be seen as consuming part of a *global privacy budget* that should not be exceeded. DP is *mature algorithmically*: most popular ML algorithms have been adapted to *individually* enforce the DP guarantee. There are also libraries that implement these algorithms, including TensorFlow Privacy [21], Opacus for PyTorch [18], and multiple libraries for statistics [20, 29, 47].

Comparatively, DP research is *primitive on systems* that enforce a global DP guarantee across *multiple* DP algorithms. Indeed, enforcing a global DP guarantee creates scheduling challenges that have never been addressed in the literature. For example, given a dynamic ML workload of multiple models trained on the same user data stream, how should the global privacy budget be allocated to maximize the number of models that are successfully trained with DP? Recently, we presented Sage, an incipient design of an ML training platform that maintains a global DP guarantee for a dynamic workload of ML pipelines operating on a continuous data stream [35]. Our key contribution was to show that by splitting the data stream into *blocks* (for example by time), enforcing a global DP guarantee over the entire stream reduces to enforcing the guarantee on each block. This showed at a basic level how to operationalize a global DP guarantee for a dynamic ML workload. but left the challenging questions related to scheduling unresolved. Moreover, our block notion was rudimentary, supporting only limited DP semantics (Event DP, which offers non-ideal protection [33, 41]) and basic DP composition methods (which scale poorly with the number of models).

In this paper, we present *PrivateKube*, a plug-in extension to the popular Kubernetes workload orchestrator that can be used to schedule global privacy budgets for a dynamic workload of DP ML pipelines akin to Sage's. The key insight is to (1) generalize the notion of private blocks to support a

[*] First co-authors of the paper with equal, complementary contributions.

wider range of DP semantics and composition methods, and (2) incorporate private blocks as *a new, native resource* into Kubernetes, alongside traditional compute resources (such as CPU, GPU, and RAM), so they can be scheduled uniformly. Despite intuitive correspondence of our privacy abstraction to Kubernetes abstractions for traditional resources, there are also significant semantic differences that force us to redesign the scheduling at a fundamental, algorithmic level.

Specifically, private blocks differ from traditional computing resources in two key dimensions. First, once a portion of a private block is allocated to a task, it can never be recuperated. Second, in many use cases, the utility of using private blocks is a step function: if a task has enough privacy budget it can make progress, but if it does not have sufficient budget, its accuracy can be affected in complex ways and it is often preferable to wait to accumulate enough budget before proceeding. These two properties invalidate assumptions typically made by scheduling algorithms for traditional computing resources, such as the popular DRF [19], which we show loses the max-min fairness property if applied directly to private blocks. In fact, we find that the very definitions of standard game-theoretical scheduler properties require change to apply to the characteristics of the privacy resource.

We develop a new algorithm for scheduling private blocks, called DPF (Dominant Private block Fairness). DPF treats each private block as a *separate resource* that can be demanded (or not) by tasks. Different tasks can demand different private blocks, creating heterogeneous resource demands and pointing to multi-resource scheduling algorithms, such as DRF [19], as a basis for DPF. Similar to DRF, DPF allocates private blocks to the user that has the minimal *dominant private block share* – the maximum privacy budget requested by a user across the private blocks. Different from DRF and other related scheduling algorithms [32, 49], DPF releases privacy budgets progressively into the blocks, to ensure that future pipelines have access to the privacy resource in accordance to a fairness policy. Moreover, DPF allocates requested budgets all-or-nothing to ensure that pipelines can achieve their accuracy goals. We prove that DPF satisfies several important game-theoretic properties: sharing incentive, strategy-proofness, dynamic envy-freedom (a variant of traditional envy-freedom), and Pareto efficiency.

We evaluate PrivateKube on microbenchmarks and a workload on Amazon Reviews data. We find that: (1) DPF grants more pipelines than baseline policies at a small cost in delay; (2) stronger DP semantics (such as User DP) require more budget and data, increasing the need for judicious budget allocation as with DPF; (3) adapting DPF to Rényi DP [42], the state-of-the-art composition method, enables allocation of either many more or much larger pipelines, and (4) our native integration of the privacy resource into Kubernetes lets us easily adapt the Grafana compute resource monitor to track privacy usage on par with compute usage.

Overall, this paper is the first to pose these questions: (1) what are the characteristics of the "privacy resource" in ML workloads, (2) how should scheduling algorithms support this resource, and (3) what kinds of game-theoretical properties can be guaranteed for this resource? The answers, which form our primary contributions, are: (1) the abstraction of the privacy resource as dynamically-arriving, non-replenishable private blocks, (2) the DPF algorithm, and (3) the theoretical properties of DPF. All these are integrated into real systems, Kubernetes and Kubeflow, in a prototype that we have open-sourced: https://github.com/columbia/privatekube.

2 Threat Model and Background

2.1 Threat Model

We are concerned with the sensitive data exposure that may occur when pushing models trained over user data to untrusted locations, such as mobile devices [6,27,54], model stores that are widely shared among teams in a company [36,56], or even opened to the world via prediction APIs [2, 43]. Our focus is not on singular models, pushed once, but rather on workloads of many models, trained periodically over increasing data from user streams. For example, a company may train an auto-complete model daily or weekly to incorporate new data from an email stream, distributing the updated models to mobile devices for fast predictions. Moreover, the company may use the same email stream to periodically train and disseminate multiple types of models, for example for recommendations, spam detection, and ad targeting. This creates ample opportunity for an adversary to collect models and perform *privacy attacks* to siphon personal data.

Two classes of privacy attacks are particularly relevant: (1) *membership inference*, in which the adversary infers whether a particular entry (e.g., user) is in the training set based on either white-box or black-box access to the model and/or predictions [4, 17, 28, 57]; and (2) *reconstruction attacks*, in which the adversary infers unknown sensitive attributes about entries in the training set based on similar white-box or black-box access [8, 14, 16]. We aim to ensure that an entry's *participation* in a company's model *does not increase the risk* of an adversary learning something about that entry.

Of particular concern are attacks that can access *multiple* models or statistics trained on the same or overlapping portions of a data stream. While individually these may leak limited information about specific entries, together they may leak significant information, especially when combined with side information about an entry. Consider two statistics: (1) average value of a sensitive column s (say representing user salary); and (2) average value of column s across entries whose ID differs from "1234." Individually, they reveal nothing specific about any entry in a dataset. Together, they reveal the value of sensitive column s for entry "1234." This is a trivialized example in which the queries are ideally chosen and the adversary has access to ideal side-information about their target: the ID. However, research in more practical settings has shown that releasing multiple (versions of) ML models trained over overlapping datasets increases the attacker's membership inference power compared to releasing just one [61]. Moreover, many pieces of information, such as demographic traits and locations, can be pieced together to uniquely identify individuals and used as side information in such attacks [4, 12, 45]. Thus, a significant data exposure threat stems from the repeated release of models/statistics from overlapping portions of a stream.

2.2 Differential Privacy

DP is known to address the preceding attacks [8, 16, 31, 57]. At a high level, membership and reconstruction attacks work by finding data points (which can range from individual events to entire users) that make the observed model more likely: if those points were in the training set, the likelihood of the observed output increases. DP prevents these attacks by ensuring that no specific data point can drastically increase the likelihood of the model outputted by the training procedure.

To prevent such information leakage, DP introduces *randomness* into the computation to hide details of individual entries. A randomized algorithm $\mathcal{Q} : \mathcal{D} \to \mathcal{V}$ is (ε, δ)-DP if for any neighboring datasets $\mathcal{D}, \mathcal{D}'$ that differ in one row and for any $\mathcal{S} \subseteq \mathcal{V}$, we have: $P(\mathcal{Q}(\mathcal{D}) \in \mathcal{S}) \leq e^{\varepsilon} P(\mathcal{Q}(\mathcal{D}') \in \mathcal{S}) + \delta$. Parameters $\varepsilon > 0$ and $\delta \in [0,1]$ quantify the strength of the privacy guarantee: small values imply that one draw from such an algorithm's output gives little information about whether it ran on D or D'. The *privacy budget* ε upper bounds an (ε, δ)-DP computation's privacy loss with probability (1-δ).

A key strength of DP is its *composition* property, which in its basic form, states that the process of running an $(\varepsilon_1, \delta_1)$-DP and an $(\varepsilon_2, \delta_2)$-DP computation on the same dataset is $(\varepsilon_1 + \varepsilon_2, \delta_1 + \delta_2)$-DP. Therefore, privacy loss accumulates linearly with the privacy loss of each computation. Composition lets one account for the privacy loss resulting from a sequence of DP-computed outputs, such as the release of multiple models. It is thus critical for enforcing a global DP guarantee. There are more advanced forms of composition, such as Rényi DP [42], which permit much tighter analysis of cumulative privacy loss (sublinear). We discuss those in the latter parts of the paper, because they are vital to a well-performing globally DP system, but for the next two sections we assume basic composition for simplicity.

Multiple DP mechanisms exist, such as the Laplace and Gaussian mechanisms. They add noise to the computation from a Laplace/Gaussian distribution scaled by a function of ε, δ, and the sensitivity of the computation. The noise scale depends linearly in $1/\varepsilon$ and at most logarithmically in $1/\delta$. When enforcing a global DP guarantee, which we denote in this paper as $(\varepsilon^G, \delta^G)$, both parameters become "resources" that must be allocated among the individual computations to ensure that cumulatively the computations do not exceed either. However, because individual computations are much more sensitive to the allocated ε than to δ, throughout this paper we will focus on ε^G as the sole global resource to schedule. In evaluation, we set the individual δ requested by each pipeline small enough in comparison to δ^G (10^{-9} and 10^{-7}, respectively) such that ε^G is always the bottleneck.

The DP semantic can be instantiated at multiple granularities, the difference being what a "row" corresponds to. *Event DP* enforces DP on individual data points (e.g., individual clicks). *User DP* enforces DP on all data points contributed by a user. It is stronger but challenging to sustain when new models must keep training on new data from the same users. *User-Time DP* is a middle-ground that enforces DP on all data points contributed by a user in a given period (e.g., one day).

2.3 Assumptions

Our overarching goal is to *develop infrastructural support for organizations to enforce a global DP guarantee – at Event, User, or User-Time level – across the entire ML workload they operate on sensitive data streams*. This would let organizations control the leakage of personal information through the models. The focus of this paper is on how to orchestrate the global privacy budget across *competing* but *trusted* ML training processes, each of which is assumed to be coded by their programmers to enforce DP. We assume that the programmers are trusted to correctly implement DP training processes and to adhere to the protocols we establish for them. Moreover, we assume that the training processes themselves, plus the compute infrastructure, are trusted. For example, if our scheduler refuses to allocate a requested privacy budget to a training task, the task will not access the data. If the scheduler allocates the task's requested budget, ε, then the training process will not attempt to use more than ε. On the other hand, programmers may be incentivised to achieve higher accuracy for their models by requesting more ε. Therefore, we must provide users with strong incentives to fairly share ε^G.

3 PrivateKube Architecture

PrivateKube is a plug-in extension to the popular Kubernetes workload orchestrator. It can be used to allocate privacy budgets for a dynamic workload of ML pipelines to enforce a global $(\varepsilon^G, \delta^G)$ DP semantic. Our key insight is to incorporate the privacy budget as *a new, native resource* alongside traditional compute resources so developers can manage compute and privacy uniformly. Despite one-to-one correspondence of our privacy resource abstractions to traditional Kubernetes abstractions, there are also significant semantic differences that cause us to re-think scheduling for the privacy resource. This section gives an architectural view of our privacy resource abstraction, with the similarities and differences from Kubernetes' abstractions. §4 then describes *DPF*, the first scheduling algorithm suitable for the privacy resource. §5 presents extensions of DPF to support both Rényi composition and all three DP semantics: Event, User, User-Time. These, too, constitute firsts for the DP systems literature.

3.1 Overview

Fig. 1 shows the PrivateKube architecture alongside the main components of a standard Kubernetes deployment. It

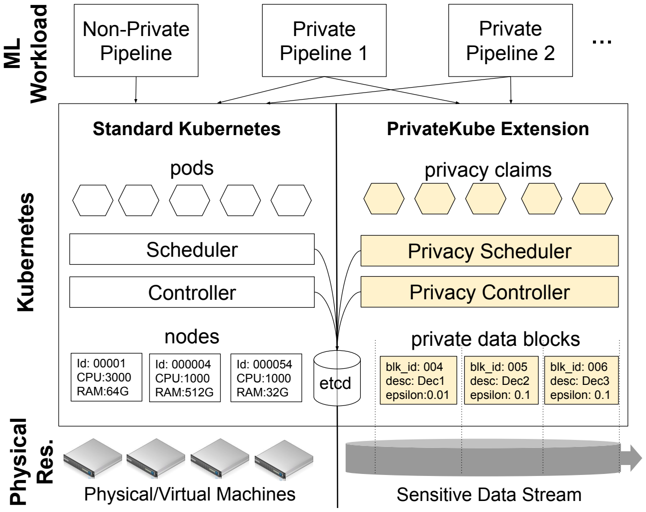
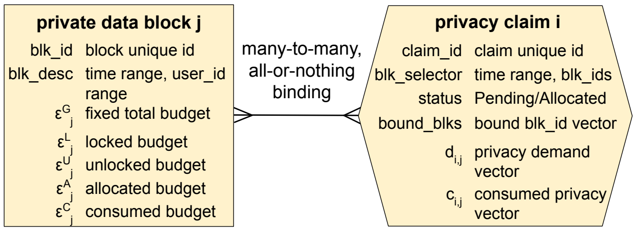


Fig. 1: **PrivateKube architecture.** Clear components are standard Kubernetes. Highlighted components (yellow) are added by PrivateKube.



blk_id	block unique id
blk_desc	time range, user_id range
ε^G_j	fixed total budget
ε^L_j	locked budget
ε^U_j	unlocked budget
ε^A_j	allocated budget
ε^C_j	consumed budget

claim_id	claim unique id
blk_selector	time range, blk_ids
status	Pending/Allocated
bound_blks	bound blk_id vector
$d_{i,j}$	privacy demand vector
$c_{i,j}$	consumed privacy vector

API: allocate(claim_id, blk_selector, $d_{i,j}$) → Success/Failure
consume(claim_id, $c_{i,j}$) → Success/Failure
release(claim_id)

Fig. 2: **PrivateKube abstractions and API.** Some variables are indexed by block (j) or claim (i) for consistency with notation needed in §4.

underscores the correspondence between traditional and privacy abstractions. Kubernetes orchestrates the execution of a *workload* – in our case an *ML workload* consisting of multiple training pipelines – onto the *physical resources* available to the Kubernetes deployment. In standard Kubernetes, the physical resources are physical or virtual machines. The main abstractions that standard Kubernetes provides are: (1) *node*, an abstract representation for a physical or virtual machine; and (2) *pod*, a containerized unit of execution. A pod specifies the container image to execute, plus the type and quantity of compute resources it demands, such as CPU, GPU, RAM, SSD. A node specifies the type and quantity of compute resources it has available. The primary functions of Kubernetes are to: (i) monitor for pods with unsatisfied resource demands (component *Controller* in Fig. 1) and (ii) *bind* each pod to one node that has the demanded resources (component *Scheduler*). Once a pod is bound to a node, the pod's image is executed.

PrivateKube extends Kubernetes to add a new type of physical resource: sensitive data streams. We correspondingly add two new abstractions to Kubernetes: (1) *private data block* and (2) *privacy claim*. Private data blocks (or *private blocks* for short) constitute non-overlapping portions of a sensitive data stream, such as daily windows of data from that stream. Private blocks are the finest granularity at which data can be requested by a training pipeline, and the level at which PrivateKube keeps track of the total privacy loss incurred by an ML workload of multiple pipelines. Private blocks specify the portion of the data they represent (e.g., the start and end times of the corresponding window), plus the privacy budget still available for use in that window. *Privacy claims* are used by training pipelines to demand privacy budget for the private blocks they are interested in. A pipeline specifies in its privacy claims a selector for the private blocks it is requesting (such as the window of time from which they want data), plus the privacy budget it demands for these blocks. The primary functions of PrivateKube are to: (i) monitor for privacy claims with unsatisfied private block demands (component *Privacy Controller* in Fig. 1) and (ii) *bind* each privacy claim to the private blocks it demands (component *Privacy Scheduler*).

In a Kubernetes deployment with PrivateKube enabled, the workload may consist of a mix of non-private pipelines (which interact with insensitive data) and private pipelines (which interact with sensitive data). Each pipeline has multiple steps organized in a directed acyclic graph, including steps that read the data, transform it, train models, etc. The non-private pipeline interacts with standard Kubernetes to schedule its steps for execution by registering a pod for each step as soon as the step's inputs are available. The private pipeline interacts not only with standard Kubernetes (to allocate compute resources for each step) but also with PrivateKube (to allocate and consume privacy budget needed to execute the steps on the sensitive data in a privacy preserving way).

3.2 PrivateKube Abstractions

PrivateKube's abstractions are implemented *natively* in Kubernetes using its Custom Resource Definition extension API. Fig. 2 shows the state maintained for each abstraction. As with standard abstractions, state for custom resources is stored in the fault-tolerant, strongly consistent etcd store.

Private Block (Fig. 2, left): This abstraction has three constant fields: a globally unique block id (`blk_id`), a descriptor specifying the portion of the sensitive data stream it represents (`blk_desc`), and the global privacy guarantee PrivateKube is configured to enforce against the entire stream ($\varepsilon^G_j = \varepsilon^G$). PrivateKube supports multiple ways of splitting the stream into private blocks, and splitting determines the type of DP guarantee PrivateKube enforces: Event, User, or User-Time DP. §5 shows how splitting works for each.

Each block j also maintains four variable fields. (1) ε^C_j denotes the budget that has been consumed for the block. We leverage the theory we developed for Sage [35] to justify that enforcing a global ε^G privacy guarantee over the entire stream reduces to ensuring that $\varepsilon^C_j \leq \varepsilon^G_j = \varepsilon^G$ for all blocks j at all times. Thus, when ε^C_j reaches ε^G, we remove private block j from Kubernetes and it no longer represents a resource. (2) ε^A_j denotes the part of block j's budget that has been allocated to some claims but not yet consumed. (3) ε^U_j, called *unlocked budget*, is the unallocated and unconsumed budget made presently available for allocation to privacy claims.

(4) ε_j^L, called *locked budget*, is the unconsumed and unallocated budget not yet made available for allocation. Our DPF algorithm (§4) leverages the last two fields to unlock budget from ε_j^G progressively to ensure that future pipelines have access to the privacy resource in accordance to a fairness policy. Among all fields, the invariant is: $\varepsilon_j^G = \varepsilon_j^L + \varepsilon_j^U + \varepsilon_j^A + \varepsilon_j^C$.

Privacy Claim (Fig. 2, right): This abstraction is used by pipelines to allocate and consume privacy budget from one or more private blocks. When creating a privacy claim, the programmer specifies a selector for the data blocks relevant for their pipeline (`blk_selector`). Typically, this means specifying a time range from which the programmer wishes to obtain data samples (e.g., the past year). PrivateKube then maps this descriptor onto the private blocks that contain data samples from that time range. In addition to the block selector, the programmer also specifies the demanded privacy budget for each of the blocks that match the selector. While often the demanded privacy budget will be uniform across all selected blocks, we allow the programmer to specify a *demand vector*, $d_{i,j}$, with one separate entry for each selected block.

API (Fig. 2, bottom): We implement three functions on privacy claims: `allocate`, `consume`, and `release`. A pipeline can invoke them multiple times on the same claim, and they will be executed sequentially. `allocate` invokes the Privacy Scheduler to allocate privacy demand, $d_{i,j}$, to blocks that match the `blk_selector`. The scheduler will perform the selection, verify that every matching block has sufficient unconsumed and unallocated budget to potentially honor $d_{i,j}$, and if so, binds the matching blocks to the claim. It then adds the claim to its internal list of claims to schedule with the DPF algorithm. The scheduler will ultimately decide to allocate the request, or not. If it does, `allocate` succeeds and the caller is guaranteed that the entire demand vector $d_{i,j}$ has been allocated to the bound blocks. If it does not, the blocks are unbound, and the caller can assume that none of the requested budgets in its demand vector were allocated. `consume` invokes the Privacy Controller to deduct a part of previously allocated budget, $c_{i,j}$, from blocks already bound to the claim. The function is similarly not guaranteed to succeed, for example if the caller is asking to consume more than the budget it has left for a block. `release` invokes the Privacy Controller to reclaim a previous unconsumed allocation to a claim. For example, a pipeline invokes `release` if it decides to stop early and not execute some steps. The Privacy Controller can also invoke `release` if the pipeline that owns the claim fails.

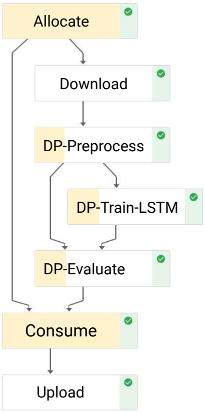
3.3 Example Pipeline

To exemplify usage of PrivateKube's abstractions and API, we describe a pipeline from our evaluation (Product/LSTM in §6.2). It is built in Kubeflow, an ML pipeline orchestrator for Kubernetes, and trains an NLP model on Amazon Reviews to predict a product category. Fig. 3 shows (a) our code in Kubeflow DSL and (b) the pipeline's execution graph. Highlighted are the distinctions between private and non-private versions.

```
@kubeflow.dsl.pipeline
def pipeline(..., blk_selector, eps):
  allocate = privatekube.allocate_op(
     blk_selector, eps)
  download = kubeflow.download_op(
     allocate.output)
  preprocess = amzn_ex.dp_preprocess(
     download.output,
     0.25*eps)
  train = amzn_ex.dp_train_lstm(
     preprocess.output,
     0.5*eps)
  evaluate = amzn_ex.dp_evaluate(
     preprocess.output,
     train.output,
     0.25*eps)
  consume = privatekube.consume_op(
     evaluate.output,
     allocate.output,
     eps)
  kubeflow.upload_op(consume.output)

kubeflow.compile(pipeline)
```

(a) **Pseudocode**



(b) **Execution Graph**

Fig. 3: **Example private Kubeflow pipeline.** Distinctions from the non-private version are highlighted in yellow background.

The pipeline has three processing steps: `Preprocess` tokenizes the reviews; `Train-LSTM` trains an LSTM model with stochastic gradient descent (SGD); `Evaluate` validates that the model passes a baseline accuracy. The Kubeflow runtime executes each step in a separate pod and passes artifacts along the computation graph [34]. If a step fails, its children in the graph will not be launched. An important note for PrivateKube is that in Kubeflow, most steps of a pipeline are pure functions and do not communicate with the outside. Only a few well-defined Kubeflow components do, including: `Download` (loads data from an external source) and `Upload` (pushes an artifact to the serving infrastructure).

Focusing on the *private version* (highlighted parts of Fig. 3), the distinctions from a non-private pipeline are two-fold. First, each step is coded by the programmer to enforce DP. For example, the training step uses DP SGD instead of SGD. The DP steps take an additional parameter: privacy budget (`eps`). The programmer splits `eps` among the steps to enforce `eps` DP at pipeline level. In the example, `dp_preprocess` gets 25% of `eps`, `dp_train` 50%, `dp_evaluate` 25% (Fig. 3a).

Second, the private pipeline interacts with PrivateKube to demand and consume `eps`. This interaction is through drop-in Kubeflow components that we created to wrap PrivateKube's API. This example highlights two such components: (1) `Allocate` and (2) `Consume`, wrappers around `allocate` and `consume`, respectively (Fig. 3b). The protocol is simple: place `Allocate` before any component accessing sensitive data (e.g., `Download`); place `Consume` before any component with externally visible side-effects (e.g., `Upload`). (1) `Allocate` creates a privacy claim and invokes `allocate` on it with a block selector and `eps` privacy budget. If `allocate` succeeds, then `Download` reads the data of the blocks bound to the claim (`bound_blks`) and the training process begins. If `allocate` fails, then `Download` is never launched and the sensitive data never accessed. (2) `Consume` receives the privacy claim from `Allocate` and invokes `consume` on it with a privacy budget equal to the one that was consumed. If `consume` succeeds, then `Upload` runs and outputs the model artifact. If `consume` fails, then `Upload` is never launched and the model never externalized. Assuming programmers adhere to this protocol (§2.3), the above ensures that PrivateKube controls the privacy loss resulting from externalizing ML artifacts.

3.4 Kubernetes – PrivateKube Distinctions

Despite one-to-one mapping of our abstractions with Kubernetes' – node::private block, pod::privacy claim – there are also semantic differences. First is the level at which we make scheduling decisions. Consider the pipeline from §3.3. The Kubernetes Scheduler performs a scheduling decision for each step. It schedules our `Allocate` and `Consume` pods, as well as the functional pods. In PrivateKube, we decided to ***allocate privacy at the level of entire pipelines***. Indeed, after being allocated compute resources, the `Allocate` pod creates a privacy claim and invokes `allocate` on it. This is when the Private Scheduler makes a scheduling decision for the privacy resource. The privacy claim is then kept for the entirety of the pipeline and passed among its components as needed.

Second, in Kubernetes, the binding of pod to node is many-to-one: one pod can be bound only to one node, but the same node can be bound to multiple pods. In PrivateKube, the binding is many-to-many: a privacy claim can be bound to many private blocks, and the same block can be bound to multiple claims. This leads to a question of atomicity for the binding across multiple blocks. A critical design decision we have made is an ***all-or-nothing semantic*** for scheduling: a pipeline can expect `allocate` on its privacy claim to either fail or guarantee that (1) all the blocks matching the claim's selector were bound to the privacy claim, and (2) for each block, the demanded privacy budget was allocated in full. This decision, which has significant impact on the scheduling algorithm (§4), should be thought of as a plausible assumption, though not the only reasonable one. Multiple use cases justify all-or-nothing. Many DP algorithms have complex interactions with hyper-parameters, such as learning rate and batch size; programmers may want to run on the budget for which those were tuned. Other use cases include the need for comparable models and DP budget searches on a fixed schedule (as proposed in Sage [35]). Furthermore, the ***non-replenishable*** nature of the privacy budget suggests that the scheduler should grant no more budget than a pipeline demanded, to keep as much budget available for future pipelines.

4 DPF Algorithm

Given the preceding integration of private blocks as a new resource in Kubernetes, we now explore how scheduling should work for this resource. Can we achieve for privacy the same types of theoretical guarantees that compute schedulers often achieve? How should scheduling algorithms change given the semantic differences between privacy and compute resources? To obtain initial answers, we focus on max-min fairness guarantees and algorithms that support them.

Our idea is to model each private block as a *separate resource* that must be allocated to different pipelines based on their demands. Demands will differ across pipelines, both in the blocks they select and in the privacy budgets they request for selected blocks. Consider four blocks $(B0, B1, B2, B3)$ and three pipelines requesting: $d_1 = (0.5, 0.5, 0.5, 0.0)$; $d_2 = (0.0, 0.1, 0.1, 0.1)$; and $d_3 = (0.0, 0.0, 0.0, 0.01)$. The pipelines could be: a large model (user embedding) registered before block B3 appeared; a smaller model that needs recent data (news recommendation) registered after B3 appeared; and a daily statistic invoked on B3. Privacy demands being heterogeneous, the four blocks will have heterogeneous capacities left after the pipelines complete.

The preceding formulation points to DRF (Dominant Resource Fairness) [19] – an algorithm that achieves max-min fairness for multiple, heterogeneous compute resources (e.g., CPU, memory) – as a basis for scheduling privacy. However, as we will show, DRF's max-min fairness guarantees do not hold for scheduling privacy. We next describe the limitations of DRF and several variations for privacy scheduling, after which we present the design and analysis of our new algorithm, *DPF* (*Dominant Private block Fairness*).

4.1 Limitations of DRF and Variations

We identify three limitations of DRF with respect to the privacy resource. First, DRF assumes static resources and sometimes even static workloads. In PrivateKube, we focus on a ***dynamic setting***: both pipelines and private blocks arrive to the system dynamically. If we applied DRF on private blocks, at every point in time, DRF would try to consume the entire available budget to satisfy the demands of all present tasks. This would make it violate the sharing incentive guarantee of max-min fairness. A new task arriving to the system that asks for its fair share of privacy budget might not be able to get it, since DRF had already allocated the budget to previous tasks.

Second, DRF, like most scheduling algorithms for compute resources [9, 23–25, 58], assumes these resources are *replenishable*: a resource can grant utility (i.e. via CPU cycles, network bandwidth) indefinitely. For instance, if multiple pipelines need to time-share a CPU core, prior work assumes that if a pipeline was assigned to the core in time interval T_1, the core will naturally be available for other pipeline in time intervals T_2, T_3, etc. and provide them with the same amount of CPU cycles per time slot. In contrast, an individual private block is a ***non-replenishable resource***. If a pipeline is assigned a budget for a particular private block, that budget is consumed forever, and there may not be sufficient budget remaining for another pipeline in that particular block. *Dynamic DRF* [32], a more recent extension of DRF, considers both dynamic settings and non-replenishable resources. Unfortunately, Dynamic DRF has its own limitation, as follows.

Third, as discussed in §3.4, PrivateKube adopts an ***all-or-nothing semantic***: a pipeline is either allocated all of its demanded budget, or none at all. Therefore, pipelines have an all-or-nothing utility function, where they can only be scheduled (with a utility of 1) if *their entire demand vector is allocated*, otherwise their utility is 0. Once a pipeline is allocated its entire demand vector, it leaves the system. Having an all-or-nothing utility function departs from both Dynamic DRF and DRF, which assume compute resources with continuous utility. In fact, an all-or-nothing utility function would break the Pareto efficiency of Dynamic DRF and DRF alike, which allocate resources proportionally based on demand (see §7).

4.2 DPF

Due to the ***dynamic*** arrival of pipelines and the ***non-replenishable*** nature of private blocks, we need to *gradually unlock* privacy budget as pipelines arrive to the system, in order to award those pipelines their fair share. Therefore, we need to define a more constrained notion of a *fair share* that divides the budget of private blocks over some particular number of pipelines, or a particular time period. This section presents a version of DPF that defines a fair share over the *first N pipelines that select particular private blocks*, and provides formal fairness guarantees *for those first N pipelines*. For any subsequent pipelines (after the first N) that request a budget for those particular blocks, PrivateKube will *not* guarantee them a fair share, but will make a best-effort to schedule them with leftover budget. §5 discusses a version of DPF that instead of dividing resources by pipelines, divides resources by time intervals, and has weaker fairness guarantees. In both cases we ensure that DPF schedules budget ***all-or-nothing***, so that no budget is wasted on tasks that will not end up being scheduled, thus violating Pareto efficiency.

Algorithm 1 gives pseudocode for DPF. When a new block j is created (ONDATABLOCKCREATION), its per-block budget, ε^G_j, is determined by the fixed global privacy budget ε^G. To ensure that the first N tasks that request j get their fair share, j's budget is initially completely *locked* ($\varepsilon^U_j = 0$).

Recall that each pipeline in PrivateKube has in its privacy claim a privacy demand vector, d, whose entries represent the epsilon demand for the private blocks matching the claim's selector. We define the *privacy budget fair share* of each private block j as: $\varepsilon^{FS}_j = \varepsilon^G_j / N$. DPF guarantees the fair share of a given private block j to the first N pipelines that arrive to the system that have a non-zero demand for j.

We unlock the budget as pipelines arrive (function ONPIPELINEARRIVAL): a new pipeline i that requests budget from a particular block j unlocks ε^{FS}_j of that block's budget, up until all the block's budget is unlocked. The scheduler's responsibility is to allocate the total unlocked budget (ε^U) among the different pipelines.

To determine which pipeline gets scheduled first, the scheduler maintains a sorted list of the waiting pipelines, based on their *dominant private block share*. This is defined as the maximum demand within each pipeline's demand vector:

$$\text{DominantShare}_i = \max_j \frac{d_{i,j}}{\varepsilon^G_j}, \tag{1}$$

where $d_{i,j}$ is the demand for block j of pipeline i and ε^G_j is the total budget of private block j. The scheduler sorts pipelines by their dominant private block share, with the smallest share ranked first (function ONSCHEDULERTIMER). If there are one or more pipelines that have the same dominant private block share, DPF will sort them by taking the smallest of the second-most dominant private block share of each pipeline, followed by the smallest third-most dominant share, etc.

Algorithm 1 DPF (max-min fairness for first N pipelines).

```
# Config.: ($\varepsilon^G, \delta^G$) global DP guarantee to enforce.
function ONDATABLOCKCREATION(block index $j$)
    $\varepsilon^G_j \leftarrow \varepsilon^G, \varepsilon^U_j \leftarrow 0, \varepsilon^A_j \leftarrow 0, \varepsilon^C_j \leftarrow 0$
end function
function ONPIPELINEARRIVAL(demand vector $d_i$)
    for $\forall j : d_{i,j} > 0$ do
        $\varepsilon^U_j \leftarrow \min(\varepsilon^G_j,\ \varepsilon^U_j + \frac{\varepsilon^G_j}{N})$
    end for
end function
function ONSCHEDULERTIMER(waiting pipelines wp)
    sorted_pipelines ← sortBy(DOMINANTSHARE, wp)
    for $i$ in sorted_pipelines do
        if CANRUN($d_i$) then
            ALLOCATE($d_i$)
            Run task $i$, which either consumes $d_{i,j}$ (moving
              it to $\varepsilon^C_j$) or releases it (moving it back to $\varepsilon^U_j$).
        end if
    end for
end function
function DOMINANTSHARE(demand vector $d_i$)
    return $\max_{j:\ d_{i,j}>0}\ \frac{d_{i,j}}{\varepsilon^G_j}$
end function
function CANRUN(demand vector $d_i$)
    return $\forall j :\ d_{i,j} \leq \varepsilon^U_j$
end function
function ALLOCATE(demand vector $d_i$)
    for $\forall j$ do
        $\varepsilon^U_j \leftarrow \varepsilon^U_j - d_{i,j}$
        $\varepsilon^A_j \leftarrow \varepsilon^A_j + d_{i,j}$
    end for
end function
```

DPF tries to allocate pipelines based on their order in the list. It tries to allocate *all of the demanded privacy budget vector of the pipeline at once*. If it cannot allocate the pipeline fully (function CANRUN returns false), then it moves to the next one in the list, until it reaches the end of the list.

Example. Fig. 4 shows an example run of DPF with three pipelines and two private blocks. Suppose the fair share (ε^{FS}) of each block is equal to 1. Pipeline 1 (P_1) arrives at $t = 1$, then P_2 and P_3 at each time unit. The demand vector of P_1 is $d_1 = (0.5, 1.5)$, while the vector of P_2 is $d_2 = (1.0, 1.0)$ and P_3's demand is $d_3 = (1.5, 1.0)$. The bottom of the figure depicts the state of of DPF's sorted list at each time unit, where

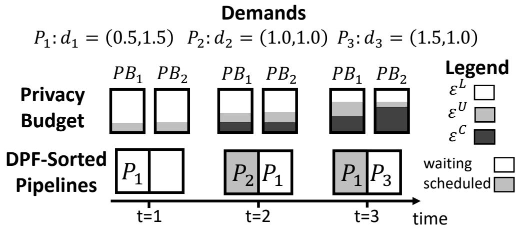


Fig. 4: **DPF example.** DPF is scheduling three pipelines (P_1,P_2,P_3) over two private blocks (PB_1, PB_2), over time. Shows the state of DPF's sorted list, and what portion of each private block is locked (ε^L), unlocked (ε^U), and consumed (ε^C). Assumes budget is consumed instantaneously ($\varepsilon^A = 0$).

the shaded pipeline in the list is the one that is scheduled at that time unit, while the unshaded one remains waiting.

When P_1 arrives it unlocks a privacy budget of 1 in each block. Since it is the only pipeline in the system (and therefore has the minimum dominant resource), the scheduler tries to allocate it a budget. However it is unable to do so, since P_1 requires a budget of 1.5 from PB_2 but only 1 is unlocked.

When P_2 arrives, more budget is unlocked. The dominant resource of P_1 is then the second block (with a demand of 1.5) and the dominant resource of P_2 is either block 1 or 2, each of which has a share of 1. Therefore, the scheduler tries to allocate budget to P_2, and does so successfully. It then tries to allocate budget to P_1, but is unable to (since there is only a budget of 1 left in PB_2). P_1 will have to keep waiting. When P_3 arrives, its dominant share is for block 1 (1.5), while the dominant share for P_1 is block 2 (1.5). Since their dominant share is the same, DPF orders them based on their second highest share, which is 0.5 for P_1 and 1.5 for P_2. Therefore, the scheduler allocates the budget for P_1. P_3 must wait, since the remaining unlocked budget for block 2 is only 0.5.

4.3 DPF Analysis

We prove four properties of DPF: *sharing incentive*, *strategy-proofness*, *dynamic envy-freeness*, and *Pareto efficiency*. We use the same definitions for these properties defined for dynamic environments based on Kash, et.al. [32].

Definition 1 (*fair demand pipeline*)**.** A fair demand pipeline has two properties: (a) the pipeline is within the first N pipelines that requested some budget for all its requested blocks, and (b) its demand for each one of the blocks is smaller or equal to the fair share (i.e. for pipeline i, $\forall j : d_{i,j} \leq \varepsilon_j^{FS}$).

Theorem 1 (*sharing incentive*)**.** A fair demand pipeline is granted immediately.

Proof. Consider a fair demand pipeline i with demand d_i. We proceed by induction over the number of waiting pipelines. *Base case:* no waiting pipelines. $d_{i,j} > 0 \Rightarrow \varepsilon_j^{FS} \leq \varepsilon_j^U$, since ε_j^{FS} is unlocked by d_i. d_i is fair so $d_{i,j} \leq \varepsilon_j^{FS} \leq \varepsilon_j^U$. The pipeline is granted, and no fair pipeline is waiting. *Induction step:* Consider any waiting pipeline k with demand d_k and dominant share DominantShare_k. By the induction assumption no fair pipeline is waiting, so $\text{DominantShare}_k > \varepsilon_j^{FS} \geq \text{DominantShare}_i$. As before, $d_{i,j} > 0 \Rightarrow d_{i,j} \leq \varepsilon_j^{FS} \leq \varepsilon_j^U$, and d_i can be granted. d_i is ordered first so it is granted. □

Theorem 2 (*strategy-proofness*)**.** A pipeline has no incentive to misreport its demand.

Proof. A pipeline has no incentive to ask for more budget than its real demand, because: (a) its utility would not increase if it obtains more budget than it needs, (b) its dominant share will be greater or equal so it can only become less likely to get scheduled. A pipeline also has no incentive to ask for less budget than its real demand, because its utility will drop to zero if it is not allocated its demanded budget. □

Theorem 3 (*dynamic envy-freeness*)**.** A pipeline present at time t cannot envy the allocation of another pipeline present at time t, except if their DominantShares are identical.

Proof. Consider pipeline i. There are two cases. Case 1: i was granted. Its utility cannot improve due to all-or-nothing utility, there is no envy. Case 2: i is waiting. Consider any pipeline j that i envies and is non identical (i and j are strictly ordered by DPF). We show by contradiction that j was granted before i entered the system. Suppose that was not the case. When j was granted: either $\text{DominantShare}_j < \text{DominantShare}_i$ and j could be granted; or $\text{DominantShare}_j > \text{DominantShare}_i$ but i could not be granted while j could. In both bases i cannot be granted from j's allocation, which would give i a utility of zero. i cannot envy j, which is a contradiction. □

Theorem 4 (*Pareto efficiency*)**.** No allocation from unlocked budget can increase a pipeline's utility without decreasing another pipeline's utility.

Proof. Consider pipeline i. If d_i was already allocated, its utility cannot improve due to all-or-nothing utility. If i is waiting, it cannot be allocated from unlocked budget as DPF grants pipelines until no pipeline can be allocated. Allocating d_i would require extra budget, which can only come from another allocated pipeline. Since each allocated pipeline has exactly its requested budget this would decrease its utility from one to zero, which is not Pareto-improving. □

4.4 Best-effort Scheduling for Higher Demands

While DPF only guarantees immediate allocation for fair demand pipelines, the algorithm has a best-effort approach to schedule pipelines that do not have a fair demand. There are two scenarios where pipelines do not have a fair demand. First, a pipeline's demand may be higher than its fair share for at least one block. From Theorem 1, fair demand pipelines always get immediately scheduled. Therefore, if there is any leftover unallocated budget after a fair demand pipeline gets scheduled, that budget can be used to schedule pipelines with higher demands. This budget will not be needed by any future fair demand pipeline, since they unlock a budget equal to the fair share. In Fig. 4, even though pipeline 1 has a higher demand than its fair share for block 1, it still gets scheduled. Second, for the same reason, DPF can safely schedule pipelines that are not among the first N to request budget from some blocks, if there is leftover unallocated budget in those blocks.

Algorithm 2 DPF-T (shows what changes in Alg. 1).

```
# Replace onPipelineArrival with:
function onPrivacyUnlockTimer(data lifetime L)
    for $\forall j$ do
        $\varepsilon^U_j \leftarrow \min(\varepsilon^G_j,\ \varepsilon^U_j + \frac{\varepsilon^G_j}{L})$
    end for
end function
```

4.5 Scheduling Compute Alongside Privacy

DPF only schedules private blocks. However, a pipeline will also need computing resource. Currently, our PrivateKube prototype implements two schedulers: the privacy scheduler (based on DPF) schedules private blocks to private pipelines. The default Kubernetes scheduler schedules traditional computing resources for non-private pipelines, and for private pipelines that have been allocated their privacy budget. DPF's game theoretic properties hold *if* the system is bottlenecked by privacy budget, rather than computing resources. We leave open the problem of scheduling privacy together with computing resources while guaranteeing game theoretic properties.

5 DPF Extensions

We have focused so far on the core version of DPF that unlocks budget based on pipeline arrival, and uses basic DP composition and Event DP. We consider three extensions of DPF to address limitations of this core version: unlocking budget by time, using a stronger DP composition (Rényi) and stronger DP semantics (User and User-Time DP).

5.1 Time-based DPF

Gradually unlocking privacy budget is key to dealing with a non-replenishable resource and a dynamic workload. The preceding DPF algorithm unlocks ε^{FS}_j for each requested block j, whenever a new pipeline arrives. We also define a version of DPF that unlocks budget *over time*, regardless of workload. Many organizations already enforce an expiration period, L, for collected data. In time-based DPF (Algorithm 2), each block gradually unlocks its budget over its lifetime L, and the fair share is defined as $\varepsilon^{FS}_j = \frac{t}{L}\varepsilon^G_j$, where t is the interval of time at which private block budgets are unlocked. The advantage of this version is the budget unlocking is predictable and independent of the pipeline arrival patterns. Moreover, by pacing budget unlocking over the data's lifetime, we ensure that the data will have DP budget remaining while still accessible.

Unfortunately, time-based DPF does not guarantee the sharing incentive. A fair-share pipeline may overlap with many other, smaller pipelines that are ordered first and consume budget when it becomes available, forcing it to wait longer than t or even never be granted.

However, the other three properties are guaranteed by this policy. We briefly sketch out the proofs for each. Strategy-proofness is guaranteed because there is no advantage in demanding more than the real demand, since the pipeline will need to wait longer for the budget to be unlocked. Envy-freeness is guaranteed for the same reason as in the base version of DPF. At any given time DPF will prioritize the pipeline with minimum dominant private block, so a pipeline with a higher dominant resource can only be scheduled earlier than another pipeline by being granted before the other pipeline arrives. Finally, Pareto efficiency is guaranteed by the combination of all-or-nothing utility and allocation.

5.2 DPF with Rényi DP

Rényi DP [42] is an alternative DP definition that is stronger than (ε,δ)-DP for $\delta \in (0,1]$ (in the sense that Rényi DP always implies (ε,δ)-DP but the converse is not true) and is weaker than $(\varepsilon,0)$-DP ($(\varepsilon,0)$-DP always implies Rényi DP). The great benefit of Rényi DP is that it permits convenient composition of multiple mechanisms that scales much better than the basic composition we have been assuming so far. We thus believe it is important for any globally DP system to support Rényi DP, and for this reason we describe our integration of it in PrivateKube. However, the definition and formulas of Rényi DP are more complex than those of (ε,δ)-DP, so we will not attempt to detail them here. Instead, we include a Rényi DP primer in our extended paper [38] and only state here a few facts needed to understand this paper.

Rényi DP Facts. As described in §2.2, DP in general upper bounds the change in the output distribution of a randomized algorithm that can be triggered by a small change in its input. Making $\delta = 0$ in the DP definition in §2.2, we see that $(\varepsilon,0)$-DP puts a *multiplicative bound* on the change in the output distribution: $\forall \mathcal{S}.\frac{P(Q(\mathcal{D})\in\mathcal{S})}{P(Q(\mathcal{D}')\in\mathcal{S})} \le e^{\varepsilon}$. (ε,δ)-DP loosens this multiplicative bound with an *additive factor*, δ. In contrast to these definitions, Rényi DP puts an upper bound on the *Rényi divergence*, a particular measure of distance between the output distributions: $\mathrm{RényiDivergence}_{\alpha}(Q(\mathcal{D}),Q(\mathcal{D}')) \le \varepsilon$. We state three facts about this distance and Rényi DP.

First, Rényi divergence is parameterized by a parameter, $\alpha > 1$, hence Rényi DP is expressed in terms of two parameters: (α,ε). Second, for every value of α, there is a direct translation from Rényi DP to (ε,δ)-DP. The formula is: $(\alpha,\varepsilon - \frac{log(1/\delta)}{\alpha-1})$-Rényi DP implies (ε,δ)-DP for any value of $\varepsilon > 0$, $\delta \in (0,1]$, and $\alpha > 1$. Also, (∞,ε)-Rényi DP is equivalent to $(\varepsilon,0)$-DP for any value of $\varepsilon > 0$. Thus, the α parameter can be seen as adding a spectrum between pure $(\varepsilon,0)$-DP and (ε,δ)-DP; from any point on that spectrum, one can reconstruct back the traditional (ε,δ)-DP guarantee. For our work, this means that PrivateKube can use Rényi DP internally while exposing the same (ε^G,δ^G)-DP guarantee externally.

Third, Rényi DP allows tighter analysis of the privacy loss from multiple mechanisms. For example, the scale of the Gaussian distribution required to achieve (ε,δ)-DP depends linearly on $1/\varepsilon$. The scale of the Gaussian required to achieve (α,ε)-Rényi DP depends on $1/\sqrt{\varepsilon}$ (and on α). In traditional DP, when composing (summing the ε's of) k Gaussian mechanisms with the same scale, σ, the composite mechanism is equivalent to a Gaussian mechanism with σ/k scale, so it's “k times less private.” But in Rényi DP, when composing the same k Gaussian mechanisms, the composite mechanism is equivalent to a Gaussian mechanism with $\sigma/\sqrt{k}$ scale, so it’s just “$\sqrt{k}$ less private.” Thus, Rényi DP scales much better in the number of computations and should enable more pipelines to share the global budget.

DPF with Rényi DP. Our goal is to take advantage of Rényi composition without sacrificing DPF’s game-theoretical properties. One option is to pick one point in the Rényi DP spectrum (one value of $\alpha > 1$) and apply DPF as is, internally using Rényi to analyze and compose privacy loss, and ultimately translating the Rényi guarantee back into traditional DP. Unfortunately, when composing multiple, heterogeneous mechanisms (think different σ for Gaussian) in Rényi DP, it is unclear *a priori* which parameter α will ultimately give the best traditional-DP guarantee; this is because both the Rényi analysis of privacy loss and the translation to traditional DP depend on α, in inverse directions (see [38] for details). In PrivateKube, we thus choose to track a set A of $\alpha > 1$ values, and to use one that ultimately gives the best traditional-DP guarantee. As the Rényi DP author shows [42], and as we observed experimentally, fine-grained choice of values is not important, so we select several values based on recommendations from [42]: $A = \{2, 3, 4, 8, ..., 32, 64\}$.

Algorithm 3 summarizes the changes DPF requires to support Rényi. For each private block, j, PrivateKube initializes a *vector of Rényi budgets*, with one entry for each value of $\alpha \in A$, based on the preceding translation formula (function ONDATABLOCKCREATION). Other privacy variables maintained in the block similarly become vectors in α (ε^U, ε^A, etc.). Moreover, a pipeline’s privacy demand also becomes a vector *for each block*: $d_{i,j}(\alpha)$. In practice, a developer will decide on the mechanism and noise scale to use (e.g. Gaussian mechanism with scale σ), based on which a library can compute the Rényi privacy demand vector for the tracked α’s. When a pipeline is allocated (function ALLOCATE), the requested budget is deducted from each block, *and for each* α.

With these changes, the question becomes how to schedule over the α vectors. One approach is to treat each $(block, \alpha)$ tuple as a separate resource. Since DPF already supports multiple resources, its game-theoretical guarantees should hold. Indeed, this is how we compute the DOMINANTSHARE under Rényi: return the maximum demand over all requested blocks and α orders. However, treating each $(block, \alpha)$ tuple as a separate resource does not work when deciding if a pipeline CANRUN. Indeed, doing so would allocate pipelines only when enough budget is unlocked for *all* α values. However, recall that in Rényi DP, *any* α with sufficient privacy budget can be translated to an ε, δ-DP guarantee. Requiring *all* to have that would just block progress until the largest α acquires sufficient budget, which removes the benefits of Rényi composition. Instead, we allow allocation of any pipeline in which each requested block has enough unlocked budget $\varepsilon_j^U(\alpha)$ for *any* α (potentially at different α across blocks).

Algorithm 3 DPF-Rényi (shows what changes in Alg. 1).

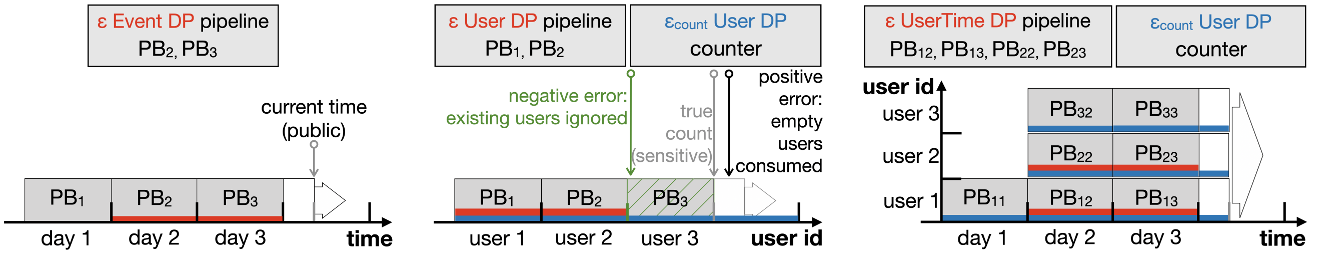
```
# Config.: $(\varepsilon^G, \delta^G)$: global DP guarantee to enforce;
#   $A$: Rényi parameters (default: $\{2, 3, 4, 8, ..., 64\}$).
function ONDATABLOCKCREATION(block index $j$)
    $\forall \alpha \in A: \ \varepsilon_j^G(\alpha) \leftarrow \varepsilon^G - \frac{\log(1/\delta^G)}{\alpha - 1}$
end function
# Either ONPIPELINEARRIVAL or ONPRIVACYUNLOCK-
# TIMER, modified to unlock budget for each alpha.
function DOMINANTSHARE(demand vector $d_i(\alpha)$)
    return $max_{j:d_{i,j}>0} \max_{\alpha \in A} \frac{d_{i,j}(\alpha)}{\varepsilon_j^G(\alpha)}$
end function
function CANRUN(demand vector $d_i(\alpha)$)
    return $\forall j: \ \exists \alpha$ s.t. $d_{i,j}(\alpha) \leq \varepsilon_j^U(\alpha)$
end function
function ALLOCATE(demand vector $d_i(\alpha)$)
    for $\forall j$ and $\forall \alpha \in A$ do
        $\varepsilon_j^U(\alpha) \leftarrow \varepsilon_j^U(\alpha) - d_{i,j}(\alpha)$
        $\varepsilon_j^A(\alpha) \leftarrow \varepsilon_j^A(\alpha) + d_{i,j}(\alpha)$
    end for
end function
```

Analysis. Under this behavior, the consumed budget at some α values may be higher than the unlocked budget, and even the global one. However, for each block j there will always remain one α such that $0 \leq \varepsilon_j^U(\alpha) \leq \varepsilon_j^G(\alpha)$. The global $(\varepsilon^G, \delta^G)$-DP guarantee is thus preserved (proof in [38]). Moreover, DPF’s four properties (§4.3) can be proven to hold under the following definition of a fair pipeline: $\forall j, d_{i,j}(\alpha) \leq \varepsilon_j^{FS}(\alpha)$, where $\varepsilon_j^{FS}(\alpha) = \frac{\varepsilon_j^G(\alpha)}{N}$ (proofs in [38]).

5.3 Supporting Varied DP Semantics

Finally, we detail how we incorporate support for all three DP semantics – Event, User, and User-Time DP – in our private block abstraction. To our knowledge, no one has shown how to support all three with one abstraction, and since we believe that a DP system should support diverse semantics, suitable for different cases, we describe here how we do so.

DP conceals a change between neighboring D and D' that are identical with a row added or removed. This neighboring definition, or what we treat as a row that is added or removed, defines the protection semantic. In *Event DP*, the most common but weakest semantic, D and D' differ in one event (e.g., one click). DP thus conceals the impact of adding or removing one such event (e.g. yesterday’s click on a health related post about a specific condition), but since one user can contribute a large number of such events, important aspects of a user’s behavior can still leak though DP computations (e.g. repeated clicks related to said medical condition). In *User DP*, the strongest semantic, neighboring datasets differ by all the data of one user. User DP conceals the entire contribution of a user regardless of the amount of data (e.g., many clicks in a health app). This semantic can be challenging to enforce on streams, since users with an exhausted privacy



(a) **Event DP.** Same as Sage [35].

(b) **User DP.** New in PrivateKube.

(c) **User-Time DP.** New in PrivateKube.

Fig. 5: **PrivateKube's support for diverse DP semantics.** Shows how the data is split into blocks and how pipelines request them. Light-gray private blocks can be requested by pipelines, white blocks are in-progress. A block's area represents its ε_j^G budget. Red portions are consumed by pipelines, blue by counters.

budget cannot contribute to new computations, even if they generate new data. *User-Time DP* is a middle-ground [33], in which neighboring datasets D and D' differ by the addition or removal of all data from one user in a given time period (e.g., one day). Repeated actions of a user in that time period are protected (e.g., a browsing session with repeated clicks), and newly generated data in the next period can still be used.

Fig. 5 illustrates how we support all three DP semantics in our private block abstraction. It requires instantiating two aspects: (1) how data is split into private blocks and (2) how blocks are requested by the pipelines.

Event DP (Fig. 5a). *(1) Splitting data:* At pre-set time intervals (e.g., a day), the data collected in this interval forms a new private block with a total of ε^G privacy budget. *(2) Requesting blocks:* Because time is public, we always know which past blocks have been created and filled with data. Pipelines registered on PrivateKube can thus request blocks from a time range of interest without risking consuming budget from an empty block. In Fig. 5a, blocks for the first three days are available. The pipeline requests data from the last two days, thereby consuming budget only for those. This design is identical to Sage [35], which supports *only* Event DP.

User DP (Fig. 5b). *(1) Splitting data:* Computing on any user's event must consume DP budget for the entire user; time-based splitting is therefore insufficient because a user's clicks can span large time intervals. Instead, PrivateKube maintains a private block for each (group of) user id(s) that will ever exist in the system, lazily instantiated. New data is added to the block responsible for the corresponding user without changing its remaining DP budget, or to a newly created block if this user is new. For instance, in Fig. 5b, only the first three users contributed data so far.

(2) Requesting blocks: This raises a challenge. Unlike in Event DP, where we know which past blocks have been created and filled with data, in User DP we do not know which users exist in the system at a given time. Knowing that would leak information about which users join when, violating User DP. Instead, PrivateKube maintains a DP counter that estimates, in a user-DP way, the number of users in the system at any time. The counter is updated periodically (e.g., daily) and consumes a bit of DP budget from every block (in blue on Fig. 5b). Since the count is noisy, pipelines requesting user blocks may sometimes overshoot and consume budget from users that do not yet exist (and therefore cannot possibly supply any data). To avoid consuming budget from empty user blocks, our design has pipelines request user blocks based on a *high probability lower-bound of the true count*. This ensures the true count is under-estimated with high probability, so no empty user is wastefully requested. Our extended paper [38] gives the specific formulas to obtain this lower bound.

The counter does consume some ε_{count}-DP budget, which is a configuration parameter of PrivateKube, fixed when PrivateKube is deployed. The budget is deducted once for each data block, upon the block's creation. For example, for Rényi-DP, ONPRIVATEBLOCKCREATION(j) initializes j's global Rényi budget vector to: $\varepsilon_j^G(\alpha) = \varepsilon^G - \frac{\log(1/\delta^G)}{\alpha-1} - 2\varepsilon_{count}^2\alpha$, where the last term corresponds to the Renyi consumption of the ε_{count}-DP counter. Since DPF always works from this $\varepsilon_j^G(\alpha)$, all DPF properties are preserved.

User-Time DP (Fig. 5c). A middle-ground between Event and User DP, User-Time DP combines both mechanisms. *(1) Splitting data:* Data is split over both user and time; newly collected data is assigned to the block managing the corresponding user and the time range that includes the data creation. Some of the blocks may be empty (e.g. user 1, day 2), but since no new data can ever be added to them once their timeframe passes, there is no cost to the future of using their DP budget now. *(2) Requesting blocks:* Blocks are requested on both time and a continuous DP counter of the number of users. The counter works similarly to User DP, except that the first (smallest time) block for a user id is created when the upper-bound of the user counter reaches this user id. This corresponds to the first time a user may have contributed data.

6 Evaluation

We implemented PrivateKube on Kubernetes 1.17. Our experiments run on Google Cloud with managed GKE on two pools of CPU (n1-standard8 machines) and GPU (n1-standard8 machines with one Tesla K80 GPU) servers. Each pool is autoscaled by Kubernetes up to a cap of 10 servers per pool.

Our evaluation seeks to answer six questions:

Q1: How does DPF compare to baseline scheduling policies?

Q2: How do workload characteristics impact DPF?

Q3: How does Rényi DP impact DPF?

Q4: How does the DP semantic impact model accuracy?

Q5: How does the DP semantic impact DPF?

Q6: Does native integration facilitate tool reuse?

We develop two methodologies. First, we create a simple, controlled *microbenchmark* that helps us explore DPF under varied workload characteristics (Q1, Q2, Q3). Second, we create a *macrobenchmark* consisting of multiple ML pipelines trained on Amazon Reviews [46] to investigate Q1, and Q4-6.

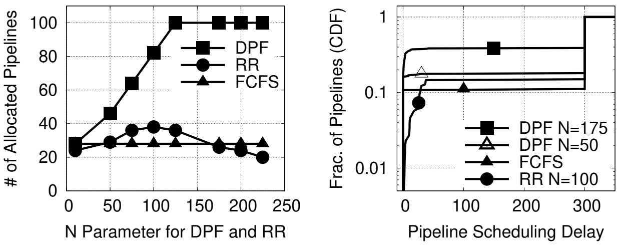
Metrics and Baselines. Across our experiments, we use the following metrics. *Number of allocated pipelines* is the number of pipelines that were successfully allocated their privacy budget throughout the experiment. *Scheduling delay* is the time measured from when a pipeline arrives to the point where it is allocated its privacy budget. *Accuracy* is the percentage of correct classification of a model.

We compare DPF to two baseline scheduling algorithms. *First-come-first-serve (FCFS)* tries to allocate pipelines by their order of arrival on available privacy budget. All the budget is immediately available to pipelines (i.e. unlocked) from the outset. *Round robin (RR)* allocates budget evenly among pipelines that are currently in the system. We implement two versions of RR that correspond to the two versions of DPF. The first one unlocks ε_j^{FS} of budget for each pipeline that arrives that demands a block j, and the second one unlocks budget in the block over time in proportion to its lifetime. For example, if the data lifetime is a year, a third of the budget of a block will be released after four months. This latter policy is similar to the one used by the Sage system [35].

Evaluation Highlights. DPF is able to grant more pipelines than the baselines at the cost of a small delay (Q1), especially over heterogeneous workloads (Q2). Rényi DP enables allocation of either many more or much larger pipelines (Q3). Stronger DP semantics require more DP budget and data (Q4), which increases the need for judicious budget allocation as with DPF (Q5). Our native integration enables reuse of existing tooling for privacy resource management, such as using Grafana to monitor privacy consumption (Q6).

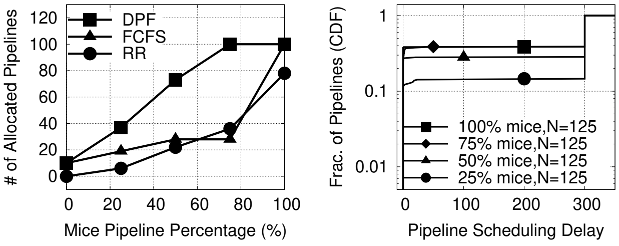
6.1 Microbenchmark (Q1, Q2, Q3)

Our microbenchmarks evaluate the performance of DPF compared to the two baselines. We assume pipeline arrival follows the Poisson process. In the single-block experiment, the pipeline arrival rate is 1 per second. We generate two types of pipelines, mice and elephants, split 75% to 25% by default, with respective demands of $\varepsilon = 0.01\varepsilon^G$, and $\varepsilon = 0.1\varepsilon^G$. In the multi-block experiment, blocks are created every 10 seconds. By default, pipeline's demand ε follows the same distribution as single-block. However, it can either request the last block with probability 0.75, or the last 10 blocks with probability 0.25, independently of the requested ε. We used a load that emphasizes the differences between the policies, where newly arrived pipelines' average demand is 13.5× of the newly generated blocks. This results in the basic composition experiments using an arrival rate of 12.8 per second, and the Rényi experiments using 234.4 per second. If not allocated, pipelines timeout after 300 seconds.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 6: **DPF behavior on a single block.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 7: **DPF with varied workload mix, single block.** (b) DPF N=125.

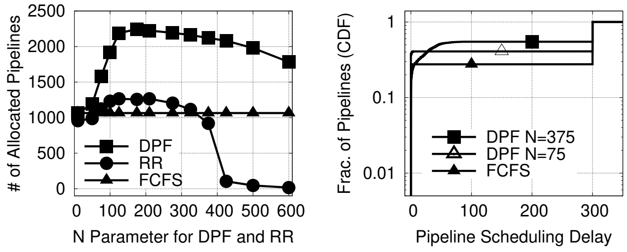
6.1.1 DPF Behavior on a Single Block

We first evaluate the performance of DPF in the simplest possible setup: with a single private block. In this case, the demand vector of each pipeline will only contain one item, and DPF will prioritize the pipeline with the lowest demand.

Fig. 6 shows DPF and RR under different N values, and FCFS. Fig. 6a shows allocated pipelines. With FCFS early elephants take away the budget of many mice, only 28 pipelines are granted. With RR, a low value of N directly unlocks all DP budget, behaving like FCFS. When N is high enough to maintain a large number of mice, but low enough to eventually grant them, RR is able to grant up to 38 pipelines (more than FCFS). At large N RR's proportional allocation creates multiple partially granted pipelines and only 20 are granted. Neither outperforms DPF. When N is equal to 1, the first pipeline unlocks all the budget and DPF behaves like FCFS. At higher values of N, DPF prefers mice over elephants and a higher number of pipelines get allocated, up to the maximum possible of 100. Since DPF never wastes budget on unallocated pipelines it outperforms RR when N is large.

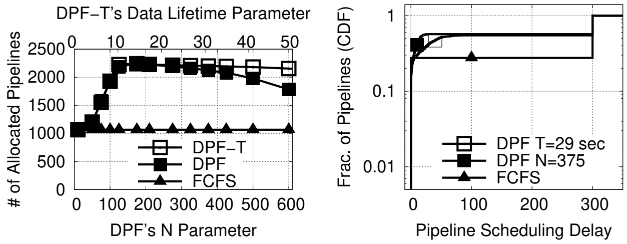
As expected, granting more jobs comes at the cost of increased delay (Fig. 6b shows scheduling delay at notable operating points for each policy). With DPF at $N = 50$ some elephants experience scheduling delays before being granted from unlocked budget. At $N = 175$ some mice wait since ε^{FS} is higher than the mice requests, but only mice are granted.

To summarize, DPF is always able to allocate budget to more pipelines than FCFS or RR. N presents a trade-off between the number of pipelines that are successfully allocated and the scheduling delay the pipelines experience.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 8: **DPF behavior on multiple blocks.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 9: **DPF and DPF-T behavior on multiple blocks.**

6.1.2 DPF Behavior with Mice Percentage

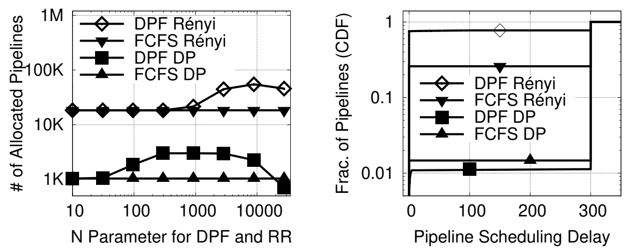
Fig. 7 compares the three scheduling policies under a variable percentage of mice and elephants. At either extreme, all pipelines are identical so DPF and FCFS allocate the same number of pipelines. In this case, the scheduling delay of FCFS is slightly better, since it always immediately schedules these pipelines. However, when there is a mix of pipelines, DPF always allocates more pipelines. RR performance is mixed: for some workloads it is able to allocate slightly more pipelines than FCFS, since it assigns a higher percentage of budget to mice; for others it underperforms FCFS, since it wastes budget on pipelines that are never scheduled.

6.1.3 DPF Behavior on Multiple Blocks

Fig. 8 shows the multi-block experiment results are similar to the single-block experiment. The main difference is that DPF performance with very large N drops, because some blocks do not see enough requests to unlock all their budget. For RR, proportional allocation helps cross-blocks pipelines to be granted (small N), yielding a small improvement over FCFS and $N = 1$ DPF. When $N > 400$, the multiple blocks create more DP budget spread over ungrantable pipelines, and there is no high allocation peak: RR grants collapse while DPF shows a 2× increase over FCFS.

6.1.4 DPF-N vs. DPF-T

Fig. 9 compares DPF-N, the version used throughout the paper, which unlocks budget based on arriving pipelines, and DPF-T, which releases budget based on time (§5). We observe that on low N and T they behave almost identically. This is because DPF-T will release budget on less queried blocks, sometimes allowing multi-block pipelines to be prematurely granted. On large N and T values DPF-T does much better, as all budget is eventually unlocked and some waiting pipelines can be granted, even when no new request is made to the blocks they demanded. Fig. 9b shows the delay for equivalent N and T values.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 10: **Traditional vs. Rényi DP, multiple blocks.** (a) Note log axes. Workload is highly amplified to saturate Rényi. (b) DPF N=8875.

6.1.5 Traditional DP vs. Rényi DP

Fig. 10 compares the DPF algorithm with traditional DP (the default DP composition used in the paper), against Rényi DP, including FCFS with both compositions as a baseline. The results show that switching to Rényi DP results in much better pipeline allocation: Rényi DP allows DPF to allocate more than 17× more pipelines than traditional DP, at their respective peaks. Even FCFS using Rényi DP significantly outperforms DPF with traditional DP. Note that DPF provides a benefit at different values of N for the two compositions, since Rényi DP requires a higher N value to reach the point where DPF starts prioritizing small pipelines. We conclude that switching to Rényi DP leads to much more efficient privacy budget utilization, regardless of the scheduling policy.

6.2 Macrobenchmark (Q1, Q4, Q5)

We use a subset of Amazon Reviews [46] in which users and products have 5 reviews or more, and keep product categories with 1M+ reviews. Each event has a review, timestamp, user, 1-5 rating, and product in one of eleven categories (e.g., books, clothing). We keep the reviews from 01-01-13 to 01-01-18, in total 43.4M reviews from 3.7M users. Tab. 1 specifies our workload: eight ML pipelines and six summary statistics pipelines. For ML, we define four types of models for each of two tasks: product classification (assigns a review to its product category) and sentiment analysis (predicts whether a review is positive). Reviews are embedded using a Wikipedia-trained GloVe [50] except for the fine-tuned BERT model. We run non-DP architecture searches for non-DP and DP pipelines on a 1% hold-out.

We set an accuracy goal for each pipeline: for summary statistics, 5% relative error; for ML models, an accuracy reachable by User DP (e.g., 60% for LSTM/Product). Each pipeline demands the minimum amount of private blocks necessary to reach its goal with $\varepsilon \in \{0.01, 0.05, 0.1\}$ ("mice," i.e. statistics) and $\varepsilon \in \{0.5, 1, 5\}$ ("elephants," i.e. ML models). The demands range from 1 to 500 private blocks. Models use $\delta = 10^{-9}$. The workload draws 75% mice and 25% elephants. Each private block holds one day of data and has $\varepsilon^G = 10$. The experiments replay 50 days of the dataset. Pipelines register

Task	Model	Architecture*	Training
Product classification	Linear	75; 100; [] 1,111 parameters	Optimizer: Adam (for DP, non-DP). DP algo: DP-SGD (Opacus). Epochs: non-DP, event/event-time DP: 15; user DP: 60. Batch: non-DP: 256; DP: $\sqrt{N}$ for N train samples (per [1]). DP clipping: flat, max norm = 1.
	FF††	60; 100; [185, 150] 48,246 parameters	
	LSTM	30; 100; [40]† 23,171 parameters	
	BERT	L 4; H 256; A 4§ 858,379 parameters	
Sentiment analysis	Linear	50; 100; [] 101 parameters	
	FF††	30; 100; [150, 110] 31,871 parameters	
	LSTM	50; 100; [40]† 22,761 parameters	
	BERT	L 4; H 256; A 4§ 855,809 parameters	
Statistics	Reviews: total #, per category #		Laplace. Bounded user contribution: 20/day, 100 in total
	Tokens: total #, avg, stdev		
	Rating: avg		

Tab. 1: **Macrobenchmark pipelines.** $*$: Architecture column: the first line, $x;y;z$, shows the input sequence length (x), embedding size (y), and the list of hidden layers' size (z). The second line shows the number of trainable parameters. ††: Fully-connected feed-forward neural network. †: The LSTM is single directional and has no dropout. §: We use a pretrained BERT model and fine-tune the last transformer layer with over 850K trainable parameters.

with PrivateKube at exponentially distributed time intervals, at a rate of 300 pipelines per day.

6.2.1 Accuracy of Individual Models with DP Semantic

Fig. 11 shows the LSTM's product classification accuracy with increasing data, with no DP and for $\varepsilon \in \{0.5, 1, 5\}$ for each DP semantic. Other pipelines show similar trends. We make two observations. First, DP semantic has a large impact on accuracy for a given DP budget and data size. As expected, Event DP, the weakest semantic, provides the highest accuracy: 73%, 72%, and 72%, for DP budgets of 5, 1, and 0.5 respectively, on 20M datapoints. The larger budgets get close to the non-DP baseline, at 77%. User DP requires larger budgets: the largest reaches 72% while the smallest yields 68%. User-time DP's behavior is closer to, but lower than, Event DP, with accuracies of 72%, 71%, and 70%.

Second, increasing data or budget improves accuracy: the DP models approach the baseline slowly, but can reach it given enough data and DP budget. The relationship between accuracy, data, and budget however is non linear. For event DP with 20M datapoints, increasing the budget from 0.5 to 5 increases accuracy from 72% to 73%, while at 2.5M datapoints the same increase goes from 68% to 71%. This relationship also depends on DP semantics, with low budget models being disproportionately impacted by smaller amounts of data and budget. For user DP for instance, the accuracies go from 68% to 72% for 20M datapoints, and from 57% to 68% for 2.5M.

6.2.2 DPF Behavior with Macrobenchmark

Fig. 12 shows the performance of *DPF with Rényi DP* under our end-to-end workload. Fig. 12a shows the number of granted pipelines under the different DP semantics. We make two observations. First, as expected stronger DP semantics require more private block and DP budget, so fewer pipelines are granted in total: event, user-time, and user DP can grant 13.8k, 10.4k, and 6.7k pipelines, respectively. Second, as before, increasing N helps DPF prioritize later mice over current elephants, increasing the total number of pipelines granted by 67% (event), 75% (user-time) and 17% (user) compared to low N and FCFS. Fig. 12b shows the scheduling delay of user DP for N values of 200 and 400. We see that increase in pipelines granted comes at a reasonable cost in delay.

Fig. 13 shows the cumulative number of incoming pipelines below a given DP size in our workload, as well as those granted under DP and Rényi DP. The DP size of a pipeline is the sum of ε-DP budget over all requested blocks, and is a measure of the total amount of budget requested by the pipeline. The Rényi DP allocates about 29% more pipelines than DP. This difference is *quantitatively* smaller than we obtained in our microbenchmark. However, there is a big *qualitative* difference that this graph also illustrates: while DP only grants mice (cumulative budget below 0.1), Rényi DP is able to also run some elephants: it grants all pipelines with a cumulative budget below 2 and some pipelines up to 10. This confirms that Rényi DP is very valuable in realistic workload settings.

6.3 Kubernetes Tool Reuse (Q6)

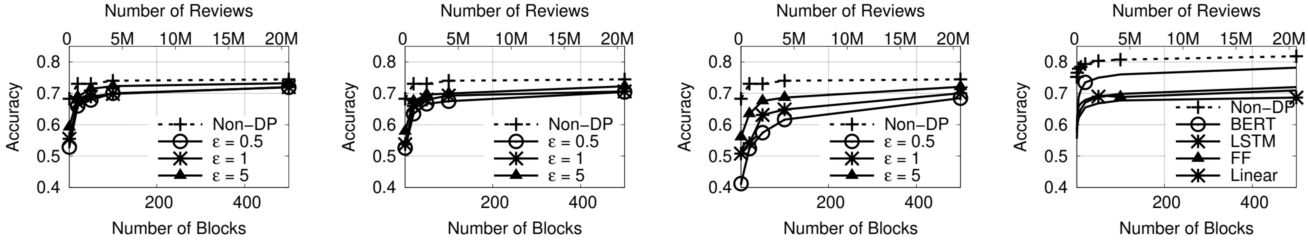
To illustrate the value of integrating with Kubernetes, we extended the Grafana-Kubernetes resource utilization monitor to track privacy usage (screenshot depicted in Fig. 14) with only 150 lines of code. We envision a suite of tools for monitoring privacy, on par with compute resources.

7 Related Work

To our knowledge, there is no work on scheduling DP, but our work builds upon a vast literature in each of these two topics.

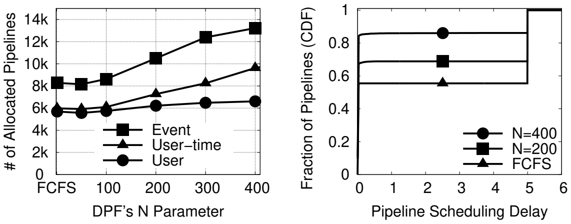
Scheduling. Decades of work exist on scheduling compute resources, such as CPU, network, memory and storage [3,7,9,10,13,19,22–25,30,37,48,49,51,53,58]. Typically, schedulers aim for max-min fairness, achieving both high system-wide utilization and high utility for each tenant. However, compute resources are replenishable, while privacy budget is not: the particular budget consumed by task i will never be available for another task in the future, whereas a CPU core granted to task i can be granted to another task after i finishes.

The two closets to our work are Dynamic DRF [32] and SEQUENTIALMINMAX [49]. Dynamic DRF provides fairness guarantees for agents arriving over time, consuming a fixed set of non-replenishable resources. Unfortunately, the all-or-nothing utility function of private blocks violates Dynamic DRF's Pareto efficiency, since Dynamic DRF would waste budget on tasks that may never get fully allocated. SEQUENTIALMINMAX is an algorithm focused on "indivisible" jobs, or jobs that have an all-or-nothing utility, and thus, similar to DPF, it only assigns resources in a sequential fashion and all-or-nothing fashion ordered by the dominant resource share. However, unlike DPF, SEQUENTIALMINMAX has static jobs, it assumes all resources are replenishable, and it does not



(a) **Product/LSTM: Event DP** (b) **Product/LSTM: User-Time DP** (c) **Product/LSTM: User DP** (d) **All Product: Event DP**

Fig. 11: **Performance of macrobenchmark Product models.** (a)-(c) Accuracy of the product classification LSTM with various DP semantics. (d) Accuracy of all four product classification models with $\varepsilon = 1$ and Event DP. The dotted baseline is non-DP BERT, whose accuracy is highest. The y axes start at 0.4, the accuracy of the naive classifier for this task (i.e. the classifier that returns the most common class).



(a) **Allocated for 3 DP semantics.** (b) **Event DP Scheduling delay.**

Fig. 12: **DPF on macrobenchmark.** $\varepsilon^G = 10$, $\delta^G = 10^{-7}$.

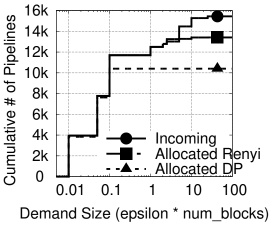


Fig. 13: **Distribution of allocated pipeline sizes.** Event DP, DPF N=400.

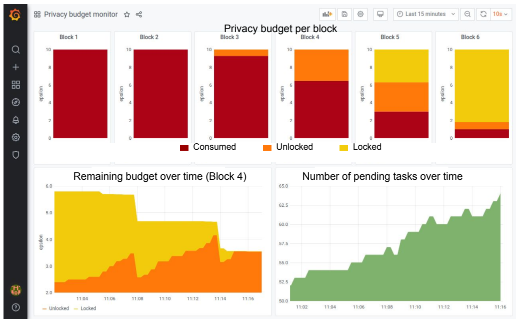


Fig. 14: **Screenshot of Grafana-Kubernetes Privacy Dashboard.**

consider dynamically arriving resources (private blocks in our case). Therefore, it provides no mechanism for gradually releasing or unlocking these resources, and would not provide a sharing incentive in our setting.

Even under a static setting, standard DRF [19] violates Pareto efficiency with all-or-nothing utility. CARBYNE schedules analytics jobs, which depend on the parallel execution of multiple tasks and have an all-or-nothing utility [24]. However, it assumes replenishable resources.

Differential privacy. There is vast literature on *DP algorithms*, which includes versions of most popular ML algorithms (e.g., SGD [1,60], Federated Learning [39]) and statistics (e.g., contingency tables [5], histograms [59]). There are also open source implementations available [18,20,21,29,47]. This literature is at a lower level than PrivateKube, and we leverage it extensively in our pipelines. Some algorithms focus on workloads [26], including on a data stream [11], but they remain very limited, supporting only linear queries.

A few *DP systems* exist, providing DP SQL-like [40,52] or MapReduce interfaces [55] to static datasets, as well as support for summary statistics [44]. None focuses on workloads of ML pipelines or supports continuous streams of data. The only such system is Sage [35], which introduces block composition for event DP, and proposes a procedure to iteratively increase a model's privacy budget until reaching a good accuracy. However, Sage does not support user and user-time DP, for which we extend block composition, and leaves the question of scheduling unexplored.

8 Conclusion

For workloads operating on sensitive user data privacy loss should be carefully orchestrated to enforce a global bound on personal data leakage. This paper presented *PrivateKube*, an extension to the Kubernetes workload orchestrator that adds differential privacy budget as a new native resource to be managed alongside traditional compute resources. PrivateKube incorporates a novel scheduling algorithm, *DPF*, the first one suitable for the unique characteristics of the privacy resource, including its all-or-nothing utility and non-replenishable nature. We show that DPF has desirable theoretical properties, outperforms baseline scheduling algorithms, and that native integration of privacy into Kubernetes can facilitate reuse of existing tools to better manage this scarce resource.

Acknowledgments

We thank Su Ji Park for developing and tuning baseline models for Amazon Reviews. We thank our shepherd, Malte Schwarzkopf, and the anonymous reviewers for the valuable comments. This work was funded by the U.S. Department of Energy (DOE) under award DE-SC-0001234; by the U.S. Army Research Office (ARO) under award W911NF-21-1-0078; by Google Research and Cloud awards; and by Sloan, Microsoft, Google, and Facebook awards.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[2] AWS. Buy and Sell Amazon SageMaker Algorithms and Models in AWS Marketplace. `https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-marketplace.html`. Accessed: 2020-12-7.

[3] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[4] Michael Backes, Pascal Berrang, Mathias Humbert, and Praveen Manoharan. Membership privacy in microRNA-based studies. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[5] Boaz Barak, Kamalika Chaudhuri, Cynthia Dwork, Satyen Kale, Frank McSherry, and Kunal Talwar. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2007.

[6] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. TFX: A Tensorflow-based production-scale machine learning platform. In *Proc. of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2017.

[7] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX Annual Technical Conference, General Track*, pages 337–352, 2005.

[8] Nicholas Carlini, Chang Liu, Ulfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. arXiv:1802.08232, 2018.

[9] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 407–424, Santa Clara, CA, March 2016. USENIX Association.

[10] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[11] Rachel Cummings, Sara Krehbiel, Kevin A Lai, and Uthaipon Tantipongpipat. Differential privacy for growing databases. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[12] Yves-Alexandre de Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific Reports*, 2013.

[13] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[14] Irit Dinur and Kobi Nissim. Revealing information while preserving privacy. In *Proc. of the International Conference on Principles of Database Systems (PODS)*, 2003.

[15] Cynthia Dwork. Differential privacy. In *Automata, languages and programming*. 2006.

[16] Cynthia Dwork, Adam Smith, Thomas Steinke, and Jonathan Ullman. Exposed! A survey of attacks on private data. *Annual Review of Statistics and Its Application*, 2017.

[17] Cynthia Dwork, Adam Smith, Thomas Steinke, Jonathan Ullman, and Salil Vadhan. Robust traceability from trace amounts. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 2015.

[18] Facebook. Opacus. `https://opacus.ai/`. Accessed: 2020-11-10.

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[20] Google. Differential Privacy. `https://github.com/google/differential-privacy/`. Accessed: 2020-11-10.

[21] Google. TensorFlow Privacy. `https://github.com/tensorflow/privacy`. Accessed: 2020-11-10.

[22] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 157–168, 1996.

[23] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery.

[24] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[25] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 81–97, USA, 2016. USENIX Association.

[26] Moritz Hardt and Guy N Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *Symposium on Foundations of Computer Science*, 2010.

[27] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. of International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[28] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V Pearson, Dietrich A Stephan, Stanley F Nelson, and David W Craig. Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays. *PLoS Genetics*, 2008.

[29] IBM. Diffprivlib. https://github.com/IBM/differential-privacy-library. Accessed: 2020-12-7.

[30] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[31] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *Proc. of USENIX Security*, 2019.

[32] Ian Kash, Ariel D Procaccia, and Nisarg Shah. No agent left behind: Dynamic fair division of multiple resources. *Journal of Artificial Intelligence Research*, 51:579–603, 2014.

[33] Daniel Kifer, Solomon Messing, Aaron Roth, Abhradeep Thakurta, and Danfeng Zhang. Guidelines for implementing and auditing differentially private systems. *ArXiv*, 2020.

[34] Kubeflow. Overview of Kubeflow Pipelines. https://www.kubeflow.org/docs/components/pipelines/overview/pipelines-overview/. Accessed: 2021-05-11.

[35] Mathias Lécuyer, Riley Spahn, Kiran Vodrahalli, Roxana Geambasu, and Daniel Hsu. Privacy Accounting and Quality Control in the Sage Differentially Private ML Platform. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[36] Li Erran Li, Eric Chen, Jeremy Hermann, Pusheng Zhang, and Luming Wang. Scaling machine learning as a service. In *Proc. of The International Conference on Predictive Applications and APIs*, 2017.

[37] Yonghe Liu and Edward Knightly. Opportunistic fair scheduling over multiple wireless channels. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, volume 2, pages 1106–1115. IEEE, 2003.

[38] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy Resource Scheduling (extended version). https://github.com/columbia/privatekube, 2021.

[39] H. Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2018.

[40] Frank D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2009.

[41] Darakhshan Mir, S Muthukrishnan, Aleksandar Nikolov, and Rebecca N Wright. Pan-private algorithms via statistics on sketches. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2011.

[42] I. Mironov. Rényi Differential Privacy. In *Computer Security Foundations Symposium (CSF)*, 2017.

[43] Model Zoo. https://modelzoo.co/. Accessed: 2020-12-7.

[44] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. GUPT: Privacy preserving data analysis made easy. In *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[45] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2008.

[46] Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP).*, pages 188–197, Hong Kong, China, November 2019. Association for Computational Linguistics. https://nijianmo.github.io/amazon/index.html.

[47] OpenDP. https://smartnoise.org/. Accessed: 2020-11-10.

[48] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 1(3):344–357, 1993.

[49] David C Parkes, Ariel D Procaccia, and Nisarg Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Transactions on Economics and Computation (TEAC)*, 3(1):1–22, 2015.

[50] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[51] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 187–198, 2012.

[52] Davide Proserpio, Sharon Goldberg, and Frank McSherry. Calibrating data to sensitivity in private data analysis: a platform for differentially-private analysis of weighted datasets. *Proc. of the International Conference on Very Large Data Bases (VLDB)*, 2014.

[53] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, March 2016. USENIX Association.

[54] Sujith Ravi. On-device machine intelligence. https://ai.googleblog.com/2017/02/on-device-machine-intelligence.html, 2017.

[55] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[56] D. Shiebler and A. Tayal. Making machine learning easy with embeddings. In *Proceedings of the Fourth Conference on Machine Learning and Systems (SysML)s*, 2018.

[57] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[58] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 349–362, 2012.

[59] Jia Xu, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, Ge Yu, and Marianne Winslett. Differentially private histogram publication. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, 2012.

[60] Lei Yu, Ling Liu, Calton Pu, Mehmet Emre Gursoy, and Stacey Truex. Differentially private model publishing for deep learning. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Santiago Zanella-Béguelin, Lukas Wutschitz, Shruti Tople, Victor Rühle, Andrew Paverd, Olga Ohrimenko, Boris Köpf, and Marc Brockschmidt. Analyzing information leakage of updates to natural language models. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 363–375, New York, NY, USA, 2020. Association for Computing Machinery.

A Artifact Appendix

A.1 Abstract

Our open-source artifact contains the main parts of the PrivateKube system, a scheduling simulator as well as experimental setups to reproduce our evaluation results.

A.2 Scope

The artifact allows to validate the microbenchmark (Fig. 6, Fig. 7, Fig. 8, Fig. 9 and Fig. 10) and the macrobenchmark (Fig. 11 and Fig. 12).

The privacy resource implementation and the DPF scheduler can be reused on any Kubernetes cluster, as well as modified to study other aspects, such as different scheduling algorithms, or the co-scheduling of privacy budgets with computational resources.

A.3 Contents

We release the following parts of the PrivateKube system: the privacy resource implementation (for both DP and RDP); the DPF scheduler (DPF-T and DPF-N); and an example of Kubeflow pipeline using PrivateKube.

We also release the discrete-event simulator, which we leverage to study and prototype scheduling algorithms of privacy and computational resources.

We also provide command line interfaces to reproduce: the microbenchmark; the DP workloads (dataset, models and parameters) used for the macrobenchmark; and the evaluation of the DPF scheduler on the macrobenchmark workloads.

The artifact does not contain: the Grafana dashboard; data ingestion pipelines and other data management infrastructure; nor a cloud-agnostic deployment for Kubeflow pipelines. We can make these components available upon request, but at the time of this publication they are fairly specific to our Kubernetes cluster.

A.4 Hosting

The artifact is available at https://github.com/columbia/privatekube/releases/tag/v1.0.

A.5 Requirements

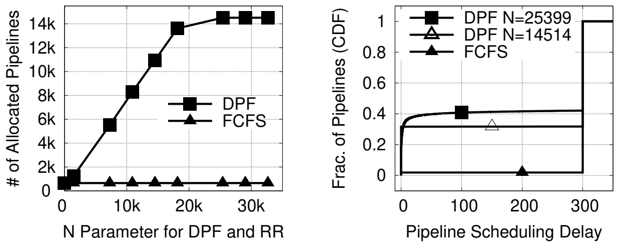
This artifact requires a Kubernetes cluster. The documentation explains how to set up a small cluster on a laptop and details the other requirements. Optionally, an NVIDIA GPU can speed up the evaluation.

The privacy resource implementation, the scheduler and the macrobenchmark do not require anything else. The Kubeflow components and the Kubeflow pipeline example require a Google Cloud Platform Kubernetes cluster with Kubeflow enabled.

It is highly recommended to reproduce the microbenchmark with a beefy machine. It normally takes us several hours to finish it with two 32-core CPUs.

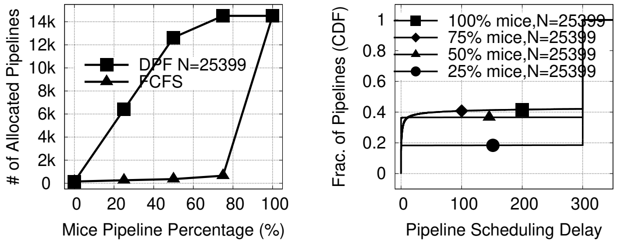
A.6 Additional Evaluation Results

The released artifact supports evaluation of PrivateKube and DPF beyond the results included in the paper. We include here a few of the results that we omitted in the paper.



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 16: **Rényi DPF behavior on a single block.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 17: **Rényi DPF behavior with variable workload mix, single block.** DPF N=25,399.

Additional Microbenchmark Results. §6.1 explores in detail the behavior of DPF with basic composition on one or multiple blocks, and under varied mice::elephant ratios. Our artifact allows exploration of these behaviors for DPF with Rényi composition, as well. For thoroughness, we include the corresponding graphs here:

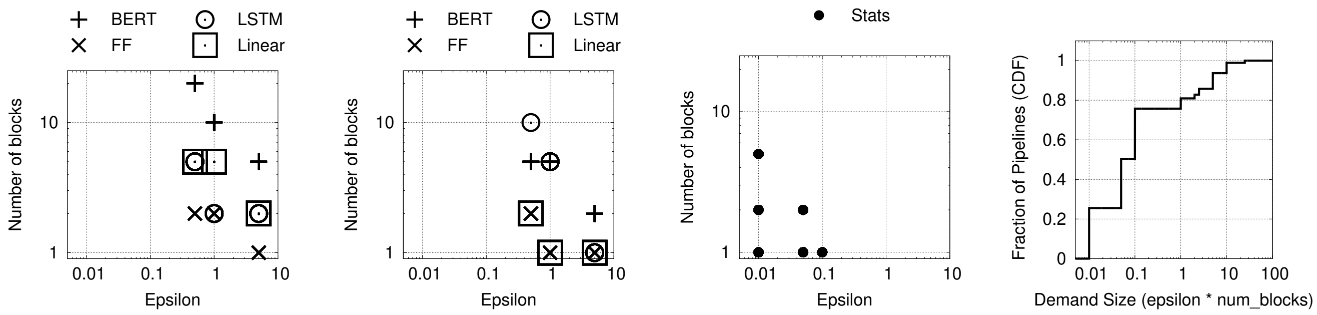
Fig. 16 (Rényi version of Fig. 6) shows that, when the load is amplified appropriately (as described in §6.1.5), Rényi DP can allocate more than 14× more pipelines than traditional DP for the optimal values of N, in the single block setting.

Fig. 17 (Rényi version of Fig. 7) shows that increasing the mice percentage has a similar impact on the number of allocated pipelines for DPF under Rényi DP and traditional DP. Similar to the basic composition results, FCFS also behaves the same as DPF when the percentage of Mice is either 0% or 100%.

Fig. 18 (Rényi version of Fig. 9) shows that, similarly to the traditional DP case, DPF performs better for large N and T. In addition, T outperforms N for large N values, since all budget is eventually locked.

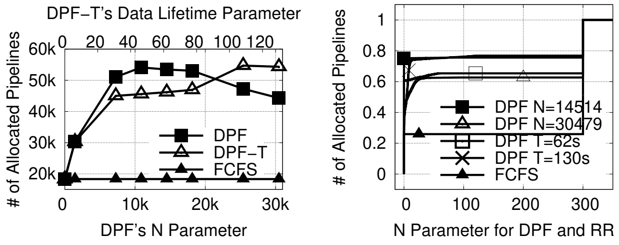
Additional Macrobenchmark Results. §6.2 shows the results from our macrobenchmark evaluation of the Rényi DP instantiation of our system. Our artifact allows evaluation of the macrobenchmark against the traditional DP instantiation as well. For completeness, we include here some of the omitted macrobenchmark results:

First, in the body of the paper, we provided an analytical description of how we chose privacy demands for our macrobenchmark workload. Fig. 15 plots the distribution of these demands for the pipelines in the Event-DP workload. The



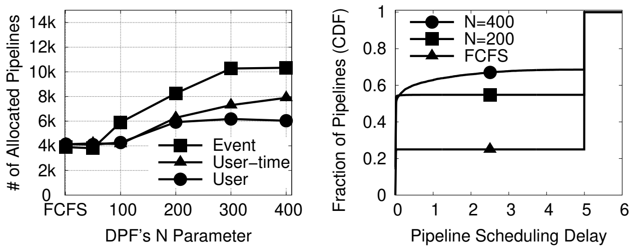
(a) **Product classification demands** (b) **Sentiment analysis demands** (c) **Statistics demands** (d) **Distribution of the demands**

Fig. 15: **Pipeline demands for the Event-DP workload.**



(a) **Number of pipelines allocated.** (b) **Scheduling delay.**

Fig. 18: **Rényi DPF and DPF-T behaviors on multiple blocks.**



(a) **Allocated for 3 DP semantics** (b) **Scheduling delay for Event DP**

Fig. 19: **DPF behavior on the macrobenchmark workload with basic composition.** The global privacy guarantee is $\varepsilon^G = 10$, $\delta^G = 10^{-7}$.

x-axis of Fig. 15a, 15b, 15c represents the ε demand in terms of traditional DP for product classification, sentiment analysis and statistics pipelines. Each ε also corresponds to the best possible DP-ε for the Rényi DP version of a given pipeline. We can see that the demands are scattered across a wide range of sizes, both in terms of blocks and epsilon, and with finer granularity than the microbenchmark's clear-cut mice and elephants. Finally, Fig. 15d shows how these varied demands are combined to form a workload. This workload gives the incoming load in Fig. 12 and Fig. 13, which evaluate PrivateKube's performance with Rényi DP.

Second, under the same workload, we add here the results from our evaluation of PrivateKube on *traditional DP* with basic composition. Fig. 19 (basic composition version of Fig. 12) shows the performance of DPF for the three DP semantics. We observe the same overall behavior as with Rényi DP: stronger semantics can allocate less pipelines, and larger values of *N* increase the number of granted pipelines. As expected, Rényi DP allocates more pipelines than traditional DP. However, as illustrated in Fig. 13, the pipelines allocated by Rényi DP are qualitatively different from the pipelines allocated by traditional DP. This effect explains why the gap in the number of allocated pipelines is smaller than in the microbenchmark, in particular when the workload contains larger pipelines (such as under User-DP).