The following appendices explore our lifetime model, algorithmic configurations, metrics, include a proof of correctness, provide additional details on simulations, and present ablation studies.

A Model Features

Our model utilizes a set of features specific to our cloud environment. Table 3 shows the list of features and a brief description of each.

Some of the categorical features, e.g., Metadata ID, VM Category, and VM Shape, take on a large range of different values. We reduce this number by collapsing any category with less than 10 examples in the training set to a catch-all "Other" category. This categorization avoids overfitting and makes the number of different categories manageable.

We analyzed the importance of different features of our model (Figure 11), using our decision forest library's *split score*, which is an indication of how much a given feature influences the model's decision. We found that the admission policy plays a major role, which identifies certain special VMs that are admitted without a quota check. We also found that the host pool and the shape of the VM play an important role in prediction.

B Model Baselines

We compare a range of different model types and libraries to identify the ideal model for our use case. We considered both survival models and regular regression models, with model structures varying across linear, tree-based and neural network-based. Among the libraries that we experiment with, Sksurv (Pölsterl, 2020) is a standard Python library for survival analysis supporting the standard linear Cox model (Cox, 2018); Lifelines (Davidson-Pilon, 2019) is a similar Python survival library allowing us to compare with a stratified Kaplan-Meier survival model (Kaplan & Meier, 1958); Xgboost (Chen & Guestrin, 2016) provides implementation of a non-linear Cox model using tree ensembles; we employ Keras (Chollet et al., 2015) and Yggdrasil Decision Forests (Guillame-Bert et al., 2023) for a standard regular neural network regression and a tree-based ensemble regression, respectively. We also experimented with different hyperparameters.

Table 4 compares models and shows that the gradient-boosted decision trees (GBDT) models perform best, achieving 99% precision at 70% recall when used to classify VMs between short and long-lived according to a 7 day threshold. We thus use these models in this paper and in production.

We experimented with regression models treating lifetimes both as linear and Log10. We found the latter to work better, since lifetimes span many orders of magnitude, which causes difficulties for regression models. We thus use Log10 for all lifetimes in our final model, including VM uptime.

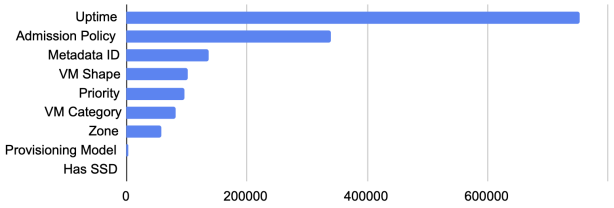

Figure 11. Impact of different features of our model on the prediction accuracy, based on the *split score* of each feature.

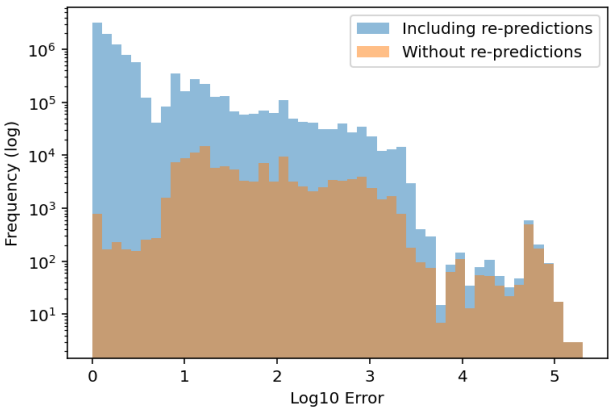

Figure 12. Histogram of the error of our GBDT model within the Log10 domain, for the first 10 million unique predictions of running NILAS with a trace.

For our production model deployments, we found that the accuracy of the model increases if we cap VM lifetimes at 168 hours (7 days). In production, all VMs with a lifetime longer than 7 days are capped. We found that capping avoids the case that a small number of very long-lived VMs "distract" the model.

For our final Yggdrasil GBDT model (Guillame-Bert et al., 2023), we use all the defaults, except for the following hyperparameters:

- Number of Trees = 2000
- Maximum Number of Nodes = 32
- Growing Strategy = Best First Global

C Model Prediction Accuracy

To better understand the performance of our model, we added instrumentation to our simulation runs that log every invocation of the model and the associated prediction. Since our traces contain the ground truth, we can then compare the prediction to the actual lifetime of the VM.

Table 3. Description of model features.

Feature	Type and Cardinality	Description
Zone	Categorical (High)	The geographical zone the VM is running in.
VM Shape	Categorical (High)	The resource dimensions associated with the VM.
VM Category	Categorical (High)	A tag indicating an internal VM categorization.
Metadata ID	Categorical (High)	An internal ID to group certain related VMs together.
Has SSD	Boolean (Low)	Does the VM have SSDs associated with it?
Provisioning Model	Boolean (Low)	Is the VM a spot instance or on-demand?
Priority	Categorical (High)	Some VMs can be pre-empted and have lower priority.
Admission Policy	Boolean (High)	Whether to admit without a quota check (used in special cases).
Uptime	Float (High)	The uptime of the VM so far, in hours (log).

Table 4. Comparison of different lifetime models.

Model	Type	Library	C-index	Precision	Recall	F1 Score
Linear Cox	survival	Sksurv	0.52	0.97	0.64	0.77
Stratified KM	survival	Lifelines	0.73	0.38	0.38	0.38
Tree-based Cox	survival	Xgboost	0.78	N/A	N/A	N/A
Neural Network	regression	Keras	0.73	0.99	0.58	0.73
Gradient-Boosted Decision Trees (GBDT)	**regression**	**Yggdrasil**	**0.84**	**0.99**	**0.70**	**0.8**

We ran with NILAS against one of our traces and recorded the first 10M predictions. Figure 12 shows the error distribution, both with and without including repredictions of the same VM. Note that both the x and y axis are log scale (i.e., a much larger fraction of the errors are within a factor of 10 – or 1 in the Log10 domain – than it might appear from the graph). Given a prediction $\hat{y}$ and ground truth y, both in seconds, we calculate the error as follows:

$$\textit{Error} = |\log_{10}(\hat{y}) - \log_{10}(y)|$$

One surprising result of this experiment was that the error distribution is not Gaussian or bimodal. Instead, we see a distribution with multiple peaks. We also observe that the distribution that includes repredictions skews significantly more to the left (i.e., lower error) than the distribution without, which confirms that repredictions improve accuracy.

D Comparing Bin Packing Metrics

Throughout the paper, we use *empty host percentage* as the main metric to capture bin packing quality. Empty host percentage is the fraction of hosts in a pool that are fully empty. The reason for this choice is that empty hosts most directly map to efficiency gains. Improving the empty host percentage by one percentage point (pp) corresponds to 1% of the pool's capacity that is now available to put in lower power mode, divest, or power down completely. Alternatively, this 1% is available to schedule large VMs that otherwise could not have been scheduled due to fragmentation. Finally, as we state in Section 4.4, there is always a small fraction of hosts that are required for maintenance and other purposes, and empty hosts make those processes faster.

Since other papers use equivalent but slightly different metrics to measure bin packing quality, we want to briefly highlight how these metrics compare:

Empty-to-Free Ratio: This alternative metric measures empty hosts. It is defined as the fraction of free CPU resources that are in completely empty hosts, containing no VMs (i.e., the number of CPU cores on empty hosts divided by the total number of free CPU cores). This metric has the advantage that it is independent of the utilization of the pool and thus helps normalize changes across pools. On the other hand, the magnitude of the resulting changes is less meaningful and we do not report them in the paper. When comparing different algorithms on the same trace, empty-to-free ratio and empty hosts behave the same, since utilization and pool size are identical between all runs.

Packing Density: Packing density is used by Barbalho et al. (Barbalho et al., 2023). It is defined as the number of allocated cores on non-empty hosts divided by the total number of cores on non-empty hosts.

In our setup, the three metrics are correlated as shown in Figure 13, since all hosts within a pool have the same number of CPUs and the composition of the pool does not change throughout the experiment. As such, improving one results in a corresponding improvement of the others.

E Analytical Proof

We prove the theorem introduced in Section 4.1 that shows an algorithm that repredicts VM lifetimes fundamentally outperforms an algorithm that does not.

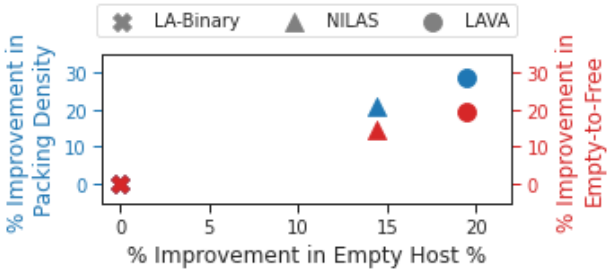

Figure 13. Comparison of different metrics from simulations done on one pool, showing relative improvements from LA-Binary.

To capture the benefit of correcting mispredictions, we consider a simplified model in which there are only two job lifetimes, which we call short (S) and long (L). We also use S and L to denote the lifetimes themselves. When a job j arrives, it has a predicted lifetime and a real lifetime. We know the predicted lifetime and we only learn the real lifetime by running the job. An algorithm assigns it to a host using some combination of a packing score and the lifetime prediction. For simplicity, we consider a best fit criteria combined with predicted lifetimes. Assuming that $S << L$ and the jobs arrive at a rate of $\lambda \geq 1/S$, the algorithm will place a job on a host with other jobs with the same predicted lifetime, thereby keeping a small number of hosts partially filled at any time. Although job sizes and the number of jobs per machine is variable, for simplicity, assume each of m hosts can hold at most k jobs.

If the predicted lifetimes are correct, this algorithm will tend to partition the jobs based on lifetime, which is basically an optimal algorithm under any reasonable set of conditions. Predictions are not perfect, but we can learn over time. We assume an error rate of ϵ, i.e., an ϵ fraction of the jobs' original lifetimes are not correct. Given two job lifetimes, once a job has run for S units of time, we learn whether it is short or long. If the algorithm then reschedules the job, we have transformed it to a perfect predictor, so we will not allow reassignment, and study the impact of having this additional knowledge.

Observe that the errors are asymmetric. If we predict that a job is long when it is short then we can recover quickly by assigning other short jobs to that machine. On the other hand, if we predict that a job is short when it is long, we now have a long job on this machine that otherwise presumably had short jobs. This case is exactly where continuous learning helps us. We classify a host as L if it has any jobs whose processing time is known to be L and S otherwise. Our algorithm will put new predicted L jobs on L hosts and new predicted S jobs on S hosts.

Now consider the effects of learning (i.e., repredictions) versus not learning. We want to compute, in an interval of time of length x, the probability that an S host has both L and S jobs. The probability that one job is mispredicted is ϵ, and in an interval of length x, we will have, in expectation, $\rho\lambda x$ L jobs, where ρ is the fraction of jobs that are L jobs. Thus the probability that there is at least one error is at least

$$1 - (1-\epsilon)^{(\rho\lambda x)} \,. \tag{1}$$

Thus, if $x \leq 1/(\epsilon\rho\lambda)$, and let let V be the event that there is an error in an interval of length X, we have

$$\begin{aligned} \Pr[V] &\geq 1 - (1-\epsilon)^{(\rho\lambda x)} \\ &\geq 1 - (1-\epsilon)^{\rho\lambda/(\epsilon\rho\lambda)} \\ &= 1 - (1-\epsilon)^{1/\epsilon} \\ &\geq 1 - 1/e \,, \end{aligned}$$

where the last inequality uses the bound $(1-1/z)^z \geq 1/e$, with $z = 1/\epsilon$.

Next, consider the case where we can learn the true values of jobs' lifetimes after they have been running for S units of time. We will show that we can now tolerate more errors, because if we learn that a host has both L and S jobs on it, we just reclassify it as a L host and continue. As the S jobs exit, we put more L jobs on the now correctly classified L host. The misclassification error does make the host unavailable for future S jobs but we still have all the other machines on which to put S jobs. We only have a problem, when we have many too many L machines and therefore, no room for the S jobs. In order to get to this position, we we will need to have cm such misclassified jobs, which will remove a constant fraction of the capacity and impact the ability to schedule S jobs.

We now show that even in an interval that is $\Omega(m)$ longer than in the previous cases, we will not have enough errors to make S jobs unscheduleable. Consider an interval whose length is $cmx/2$, which is $\Theta(m)$ longer than the interval in the no learning case. The expected number of misclassifications in such an interval is now $\epsilon\rho\lambda cmx/2 = cm/2$. But, as mentioned above, we can now tolerate cm errors. Applying standard Chernoff bounds, we see that the probability that we get cm errors (twice the expectation of $cm/2$) is small.

The above discussion for our simplified model can be condensed into the following statement:

Theorem 1. *Suppose that job lifetimes are either Short or Long, and that jobs are selected independently, to arrive at a constant rate, to be scheduled on one of m hosts, each of constant capacity. If the initial error in lifetime prediction is a positive constant, then the number of hosts required in the best fit scheduling algorithm without learning will exceed the same best fit algorithm with learning by $\Omega(m)$.*

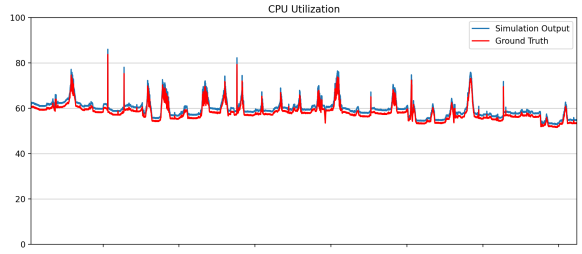

Figure 14. Validation of our simulator (y axis in %).

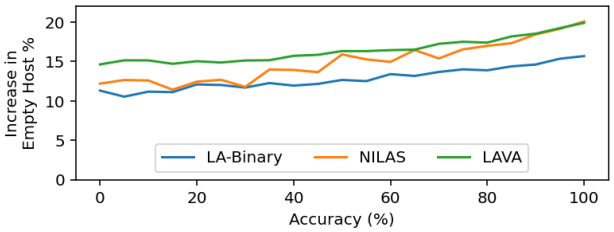

Figure 15. Performance at different levels of prediction accuracy.

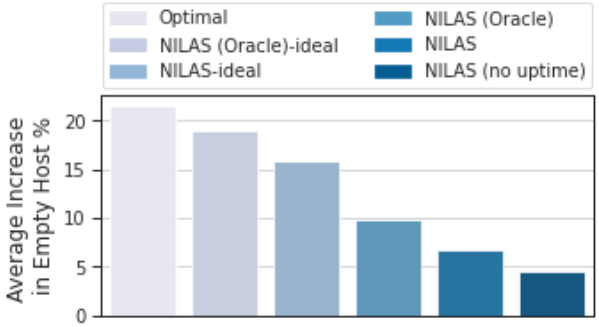

Figure 16. NILAS using oracular lifetime ran at ideal setting (cold start and highest priority) achieves near-optimal performance. NILAS also consistently outperforms the version of NILAS that does not use repredication or uptime. The reported numbers are averaged across running traces from 24 C2 pools in simulation.

F Simulator Details & Validation

We provide additional details about our time series-based simulator that we use for evaluating NILAS and LAVA.

Simulator Warm-Up: One challenge in simulating our workloads is reconstructing the exact state of the scheduler at the start of our trace, since not all system state was consistently saved. We therefore warm up our simulation by collecting all VMs that are live at the start of our trace, replay their start events in order, and then let the simulation run for another 2 simulated days to reach steady state. Warm up addresses left censorship of the trace and ensures that we run our simulations based on a snapshot that is representative of production behavior *before* lifetime-based scheduling is enabled. As we will discuss in Appendix G.2, this state is representative of a production setup where lifetime-based scheduling is enabled while the system is running. However, this methodology can reduce the impact of lifetime aware scheduling, since it takes a long time for all VMs that were placed without lifetime-based scheduling to exit.

Computing Stranding: While we directly measure the impact of our approach on empty hosts in simulation, stranding is computed via a separate pipeline (Section 2.3). We thus integrate our production stranding computation pipeline with our simulator, to compute stranding at a given point in time.

Validation: We validate our simulator by comparing its behavior to production numbers during the same time interval. We found it to be highly accurate during our validation studies (Figure 14). For example, the CPU utilization across the cluster was on average within 1.59% of the ground truth (with a standard deviation of 0.23%). We also found that the fragmentation numbers from our production experiments (Section 6.2) closely matched the simulated numbers, further supporting the validity of the simulator.

Note that the fixed offset in Figure 14 does not imply that we can easily close this gap. The offset largely stems from dynamically invested/divested capacity that can result in small fluctuations of the pool size. The offset is not always constant and can be positive or negative.

G Additional Evaluation Results

We now present additional data to further support our results in Section 6 and provide more details.

G.1 Accuracy-Performance Trade-off

We plot the trade-off curve between model accuracy and performance (Figure 15). We start from our oracular predictions and randomly categorize each VM into one of two buckets – correctly predicted or incorrectly predicted. The probability of these two cases is governed by the accuracy on the x axis. Then, we apply a different Gaussian error distribution to the label ($\sigma = 0.001$ for correctly predicted VMs and $\sigma = 3$ for incorrectly predicted VMs, in the Log10 domain). To be more representative of the actual model's behavior, we also cap lifetimes to [0, 14 days].

We see that our improvements persist with different accuracy values, and that LAVA is better than NILAS at tolerating high rates of mispredictions, as expected.

G.2 Ablations & Theoretical Limit

We next conduct a focused study of NILAS to understand how far it is from the theoretical limit, and what factors contribute to the remaining gap.

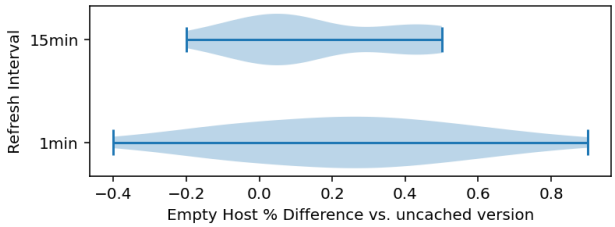

Figure 17. Effect of caching predictions, across 22 pools (in simulation). Note that repredictions are still performed when a VM is added or removed from a host, irrespective of the refresh interval.

Since all server host hardware is the same within each pool we tested, we can compute the optimal percentage of empty hosts based on the total fraction of un-reserved resources (the lower of two resource dimensions: CPU and memory) aggregated across all hosts in that pool. This optimal value sets an absolute upper bound on the total number of hosts that can be made empty under a given pool size and load.

We then test NILAS under an *ideal* simulation setting to find out the best performance it can attain. In this ideal setting, NILAS uses oracular lifetime and is put as the highest-ranked scoring function without the simulator's warm-up phase (Appendix F). Skipping warm-up amplifies the impact of NILAS since it allows NILAS to schedule VMs onto an empty cluster leveraging lifetime information throughout the whole trace, without the residual impact of VMs that were already scheduled without lifetime-awareness during the warm-up phase.

Figure 16 shows that ideal runs of NILAS get very close to the optimal result – the greedy approach taken by the NILAS algorithm is nearly optimal. However this ideal setting is unlikely to be reproduced in our production setting, where gradual roll-out (mimicked by the warm-up phase in simulation) is necessary and there are more critical scheduling criteria that rank above NILAS. These factors combined contribute to the lower performance achieved by NILAS with oracular lifetime but a non-*ideal* setting. Mispredictions from the model further bring down the amount of savings by NILAS. Additionally, we observe that not using repredictions cause more mistakes, which leads to significantly reduced performance and demonstrates the importance of our reprediction-based approach.

G.3 Ablation: Caching Predictions

As stated in Section 5, we found that repredicting every VM on each considered host becomes a bottleneck in some very large pools. We therefore introduced a caching approach. Here, we provide an ablation study to show that caching predictions and repredicting them at a coarser granularity does not adversely affect the performance of NILAS. Figure 17 shows these results: We compare simulations of our approach with caching against the baseline without cached predictions. We show two different caching intervals: 1min and 15min. We note that this interval only affects hosts that see no VM changes for extended periods of time; as mentioned in Section 5, when a VM is added to or removed from a host, the predictions on this machine are updated irrespective of the refresh interval. On average, we see a small improvement – we hypothesize that this could be due to caching addressing the "dip" in Figure 9.

H LARS Algorithm Details

This section provides additional details about defragmentation and improving it with Lifetime Aware ReScheduling (LARS).

Algorithm 1 LARS Algorithm

Input: A set of candidate hosts C for eviction
for *each candidate* $c \in C$ **do**
  v_{sorted} = Sort the existing VMs V_c on c, based on their predicted remaining lifetime, in descending order
  Send all VMs for eviction approval
  if *not all VMs are approved for eviction* **then**
    continue
  else
    Stop scheduling new VMs on c
    for $v \in v_{sorted}$ **do**
      Allocate new VM of the same shape as v
      Live migrate v to this new VM
      Once finished, remove v from c

Defragmentation is triggered when the number of empty hosts in a particular pool drops below a particular threshold. The defragmenter picks a set of candidate hosts C based on factors such as their current occupancy. It then evacuates the VMs on each host, live migrating (Ruprecht et al., 2018) them by copying them to another host. The scheduler selects the target host, using the same algorithm it uses for new VMs, but with the current VM state (e.g., lifetime prediction, resources, etc.).

We show the pseudo-code for this operation below. Once evacuation of a particular host begins, the defragmenter first confirms with a higher-level system that live migration of these VMs is consistent with system-level objectives. Once approved, the Borg stops scheduling new VMs on this host. (In the absence of live migration, this host would become empty as a function of the VMs' exit times). On each host, live migration chooses VMs to reschedule one at a time. Live migration occurs on multiple hosts concurrently, up to a configurable limit.

The first step of live migration is to choose a target host.

Live migration uses the same scheduling algorithm as Borg uses when initial scheduling a VM (e.g., NILAS, LAVA, or the original waste-minimization scheduler). Since the VMs have already been running for a period of time, live migration with NILAS and LAVA may already lead to improved placement because they take repredicted VM lifetimes into account. Once Borg allocates memory for the migrating VM on the new host, it migrates the original VM, and once migration has completed, the old instance is deleted. Once Borg migrates all the VMs, the host is empty and can be updated (in the case of system maintenance), put in low power mode, or divested/powered down.

The modification that LARS makes to this algorithm is that it performs the VM live migrations in the order of the predicted remaining lifetime. This optimization gives short-lived time to naturally exit while other VMs migrate. Each such VM saves one live migration.

I NILAS & LAVA ALGORITHM DETAILS

We provide pseudo-code for the NILAS and LAVA algorithm, to further formalize and clarify its behavior.

Algorithm 2 NILAS Scheduling Algorithm

Input: A new VM v
Output: Schedule v to a host
Find host h such that v does not exceed the largest exit time of VMs on h
if *No such h exists* **then**
  Schedule v on host h (with capacity to accommodate v) such that largest exit time of VMs on h is changed by least amount upon scheduling v on h
if *No such h exists* **then**
  Indicate v fails to be scheduled

Algorithm 3 LAVA Scheduling Algorithm

Input: A new VM v
Output: Schedule v to a host or give scheduling failure
if *there exists a recycling host h where $h.LC > v.LC$ and v can fit on h* **then**
  Let C be the set of recycling hosts h' where $h'.LC$ is closest to $v.LC$
else if *there exists a matching host h where $h.LC = v.LC$ and v can fit on h* **then**
  Let C be the set of matching hosts
else if *there exists a non-empty host h where v can fit on h* **then**
  Let C be the set of non-empty hosts
else if *there exists an empty host h where v can fit on h* **then**
  Let C be the set of empty hosts
else
  Indicate that v fails to be scheduled **return**
Schedule v to the host $h^* \in C$ selected by $NILAS$