SOL: Safe On-Node Learning in Cloud Platforms

Yawen Wang
Stanford University
Stanford, CA, USA
yawenw@stanford.edu

Daniel Crankshaw
Microsoft Research
Redmond, WA, USA
dacranks@microsoft.com

Neeraja J. Yadwadkar
University of Texas at Austin
Austin, TX, USA
neeraja@austin.utexas.edu

Daniel Berger
Microsoft Research
Redmond, WA, USA
daberg@microsoft.com

Christos Kozyrakis
Stanford University
Stanford, CA, USA
kozyraki@stanford.edu

Ricardo Bianchini
Microsoft Research
Redmond, WA, USA
ricardob@microsoft.com

ABSTRACT

Cloud platforms run many software agents on each server node. These agents manage all aspects of node operation, and in some cases frequently collect data and make decisions. Unfortunately, their behavior is typically based on pre-defined static heuristics or offline analysis; they do not leverage on-node machine learning (ML). In this paper, we first characterize the spectrum of node agents in Azure, and identify the classes of agents that are most likely to benefit from on-node ML. We then propose SOL, an extensible framework for designing ML-based agents that are safe and robust to the range of failure conditions that occur in production. SOL provides a simple API to agent developers and manages the scheduling and running of the agent-specific functions they write. We illustrate the use of SOL by implementing three ML-based agents that manage CPU cores, node power, and memory placement. Our experiments show that (1) ML substantially improves our agents, and (2) SOL ensures that agents operate safely under a variety of failure conditions. We conclude that ML-based agents show significant potential and that SOL can help build them.

CCS CONCEPTS

• **Computer systems organization** → **Other architectures**; **Cloud computing**; • **Software and its engineering** → *Software creation and management*.

KEYWORDS

Cloud computing, on-node agents, machine learning for systems, systems for machine learning.

ACM Reference Format:
Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. 2022. SOL: Safe On-Node Learning in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507704

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507704

1 INTRODUCTION

Motivation. Cloud platforms such as AWS, Azure, and GCP are complex. In addition to many control plane services running on dedicated capacity (*e.g.,* [9, 15, 36]), these platforms run many management "agents" on each server node alongside customer workloads. The agents are responsible for configuring and upgrading node software and firmware, creating and destroying virtual machines (VMs), managing resource allocation and assignment (*e.g.,* [20, 37]), checking for failure or vulnerability conditions (watchdogs), monitoring resource health, collecting telemetry, and many other tasks.

These tasks cannot be performed effectively from outside a node. For example, resource assignment must be fast (order of milliseconds) to prevent performance loss, power management (*e.g.,* capping) must change hardware settings, watchdogs need finer-grained telemetry than what is unavailable off node, and so on. Because agents compete for resources with customer workloads, platforms must constrain their resource usage and/or at least partially offload them to accelerator cards, as in [22, 26].

Many agents collect data and make frequent decisions. Currently, these decisions are based on static heuristics or results of offline analysis. But ML has shown potential to improve agent behavior through workload- and hardware-aware decision-making [6, 27, 37]. For example, an agent responsible for conserving dynamic core energy can benefit from learning the impact of core frequency on the workloads' performance at each point in time. Or a watchdog agent could learn to immediately flag serious issues with the platform, while being slower for behaviors that are most likely benign.

Unfortunately, these agents cannot take full advantage of the "centralized" ML systems currently available in production, such as Resource Central [9] or TFX [5]. These systems train models offline (using data from all nodes) and serve predictions (model inference) on-demand via a REST interface. Thus, they are limited in their model update and inference frequency by the network latency and bandwidth available for management communication. In contrast, on-node agents may need to operate on large amounts of fine-grained node-local data (*e.g.,* core utilization samples collected every tens of microseconds) and/or have to make high-frequency decisions (*e.g.,* reassigning cores very few milliseconds) [37].

Challenges. We can overcome the limitations of centralized ML systems by learning online on the nodes themselves. However, deploying safe and robust learning on nodes that run (potentially sensitive) customer workloads poses challenges. First, there are

many conditions that can lead ML-based agents to compromise quality of service (QoS). ML-based agents must be robust to unforeseen problems due to the workload, the learning model, the node environment, or even all three simultaneously, without needing human intervention or communication with a centralized service.

Second, learning on individual nodes builds models online for cloud workloads whose properties are unknown in advance. This means that the models themselves cannot be fully vetted offline ahead of time. Thus, customer workloads must be protected from the decisions of bad models running in production, rather than relying on a pre-deployment process to filter out most of these inaccurate models before they ever reach customers. Possible causes for poor ML behavior include bad input data due to corrupted or improperly configured hardware or OS counters, or attempting to learn from workloads that violate modeling assumptions (such as steady-state or periodic workload behavior).

Third, even accurate models can lead to sub-optimal agent behavior in the presence of scheduling delays. When the host resources are needed for higher-priority tasks, such as virtual IO, the agent's execution may be delayed and lead it to (1) miss important data samples and/or (2) take actions based on stale data or a stale model.

Finally, customer workloads must be protected when the agent experiences silent model failures or even hard crashes as a result of external interference, unforeseen environmental conditions, or software bugs.

Our work. We first perform a comprehensive characterization of the node agents that run in Azure, and identify the classes of management tasks that are most likely to benefit from on-node learning. We find that three classes, which collectively make up 35% of all agents, can benefit from on-node learning. Watchdog agents can use ML to both increase failure detection coverage and detect problems earlier. Monitoring agents can leverage ML to adapt where and when they collect telemetry based on node activity, increasing coverage without increasing cost. Resource control agents can use ML to improve resource utilization while protecting customer workload performance.

When deploying on-node ML for these classes of agents, it is crucial to be robust to the heterogeneous and evolving cloud environment under all failure conditions. To facilitate the development and operation of robust on-node learning agents, we introduce SOL, a **S**afe **O**n-node **L**earning framework. Agent developers can use SOL for developing ML-based agents that are internally safe and robust to the range of failure conditions that can occur in production. Different agents are typically developed by different teams in large cloud platforms. SOL provides a unified interface across teams to reduce deployment complexity. Moreover, its interface allows cloud operators (*e.g.,* site reliability engineers or SREs) to safely terminate and cleanup after misbehaving agents without knowing anything about their implementation.

We design SOL as a general framework to support a variety of on-node management agents that employ learning algorithms. By abstracting out structural similarity across learning agents and common types of problems the agents face in deployment, SOL presents a simple API with two key elements. The first is a set of functions for developers to implement the four common operations for ML-based control agents: collecting data, updating the model, getting a prediction from the model, and actuating a change based on the prediction. The second element is a set of required watchdog-style safeguards. The safeguards enumerate common failure conditions that can be hard to detect and debug in production. Agent developers must use the safeguards to internally monitor different aspects of the agents, and avoid impact to customer QoS or node health when a problem is detected. The safeguards ensure that agents are safe to deploy at scale alongside customer workloads. The exact definition of *safety* varies based on the agents' purpose, but the desire to protect customer QoS and avoid wasting resources is common.

SOL schedules and runs the developer-provided functions. It also detects and informs the agent of any scheduling violations. This is critical for avoiding the use of stale predictions under highly dynamic workloads.

We demonstrate the use of SOL by implementing three ML-based agents. Each agent manages a different resource, uses a different modeling approach, and has different data and scheduling constraints. The first is a CPU overclocking agent, SmartOverclock, that uses reinforcement learning to overclock workloads only during the phases when they can benefit. The second is a CPU-harvesting agent, SmartHarvest (introduced in [37]), that predicts CPU utilization at millisecond granularity to borrow idle cores and safely return them before they are needed. The third agent, SmartMemory, monitors each VM's memory usage to detect pages that can be migrated to remote memory without much performance impact.

Results. We present a detailed experimental evaluation of our agents, first demonstrating that on-node learning significantly improves their efficacy, and then showing that agents implemented in SOL operate safely under a variety of failure conditions. As an example, SmartOverclock improves performance up to 41% while consuming 2.25x less power over a static overclocking baseline. The SOL safeguards limit the agent's power draw increase during failure conditions to 18%, while without the safeguards the same failure condition leads to a 268% power increase.

Related work. We are unaware of similar agent characterizations from commercial clouds. The prior work on using on-node learning has focused on ad-hoc resource management agents (*e.g.,* [27, 37]), and did not consider general frameworks for engineering safe ML-based agents.

Contributions. In summary, our main contributions are:

- A characterization of (1) the existing on-node agents in Azure, and (2) the challenges involved in incorporating ML into them.
- The design of an on-node framework with an extensible API and runtime system for deploying ML-based agents that are robust to a wide variety of realistic issues.
- The implementation and detailed evaluation of three agents that demonstrate substantial improvements from on-node learning, while maintaining workload QoS and node health.

2 PRODUCTION ON-NODE MANAGEMENT

Before delving deep into on-node learning and SOL, it is important to understand the spectrum of node agents in real cloud platforms, and identify those that can benefit from learning. Thus, in this section, we first overview the classes of agents in Azure and then discuss how learning can benefit a subset of the classes.

Table 1: Taxonomy of production agents. The rightmost column lists whether the class could benefit from learning.

Class	Count	Description	Examples	Benefit?
Configuration	25	Configure node HW, SW, or data	Credentials, fire walls, OS updates	No
Services	23	Long-running node services	VM creation, live migration	No
Monitoring/logging	18	Monitoring and logging node's state	CPU and OS counters, network telemetry	Yes
Watchdogs	7	Watch for problems to alert/automitigate	Disk space, intrusions, HW errors	Yes
Resource control	2	Manage resource assignments	Power capping, memory management	Yes
Access	2	Allow operators access to nodes	Filesystem access	No

Table 2: Examples of on-node learning resource control agents. Prior work has primarily focused on resource control.

Agent	Goal	Action	Frequency	Inputs	Model
SmartHarvest [37]	Harvest idle cores	Core assignment	25 ms	CPU usage	Cost-sensitive classification
Hipster [27]	Reduce power draw	Core assignment & frequency	1 s	App QoS and load	Reinforcement learning
LinnOS [16]	Improve IO perf	IO request routing/rejection	Every IO	Latencies, queue sizes	Binary classification
ESP [25]	Reduce interference	App scheduling	Every app	App run time, perf counters	Regularized regression
Overclocking §5	Improve VM perf	CPU overclocking	1s	Instructions per second	Reinforcement learning
Disaggregation §5	Migrate pages	Warm/cold page ID	100 ms	Page table scans	Multi-armed bandits

Taxonomy of node agents. Regardless of whether an agent runs on host CPUs or an offload card, it is typically a user-level process responsible for a specific, narrowly-defined task. This makes agents simpler to develop, easier to maintain, less likely to impact node performance, and less likely to affect each other in case of misbehavior. It also makes them easier to categorize. Table 1 categorizes the agents in Azure into 6 classes. There are 77 agents, but many of them run rarely. Next, we provide an overview of each class.

1. Configuration agents control aspects of the node's hardware, software, and data. They change the node state as directed by the platform's control plane and run from every 10 minutes (configure TCP) to order of months (host OS upgrades).

2. Service agents run various node services that are critical for operating the cloud environment. These services include VM lifecycle management, on-node agent creation, and security scanning and malware detection. They run throughout the lifetime of the node, at frequencies ranging from seconds to minutes.

3. Monitoring/logging agents monitor and/or log data (off the node). For logging fine-grained telemetry, they must aggregate/compress data to reduce the amount to be sent off node. They run at different frequencies from the order of seconds to tens of minutes.

4. Watchdog agents (or simply watchdogs) check for problems that either require telemetry that is only available on the node or where detecting the problem off the node would be too slow to prevent customer impact. Watchdogs run fairly frequently, on the order of seconds to minutes.

5. Resource control agents dynamically manage resources, such as CPUs, memory, and power. Though they are not numerous, they run frequently, on the order of seconds.

6. Access agents enable datacenter operators to diagnose and mitigate incidents. Some agents run continuously, while others only run when an incident requires operator involvement.

Runtime constraints on agents. Regardless of where they run, agents compete for precious resources: on host CPUs, they compete with customer workloads; on accelerator cards, they compete with other agents and data plane operations. Unconstrained agent execution may introduce interference and tail latency effects [13]. For these reasons, each agent runs under strict compute and memory constraints defined in its configuration (*e.g.,* 1% CPU and 200MB of memory for a host-CPU-based watchdog agent). Agents also run at lower priority than customer workloads and the host OS, which means that agents may get temporarily starved or throttled.

On-node learning opportunity. Today, production agents do not take advantage of on-node learning and, thus, are not as effective as they could be. Any agents that benefit from collecting data about current workload characteristics to guide dynamic adjustment of their behavior can potentially take advantage of ML. In particular, we argue that resource control, monitoring/logging, and watchdog agents can benefit significantly from on-node learning.

Resource control agents can benefit because the most efficient assignment of resources (particularly compute, memory, and power) is highly dependent on the current workload. Learning online directly on each node can predict the short-term workload dynamics, and make better assignment decisions without affecting customer QoS. Better assignments offer opportunities for improved efficiency and cost savings. It is thus the focus of our case-studies in §5. Prior work has also considered learning-based resource control agents (see Table 2).

Monitoring/logging agents can benefit because there is a cost to collecting samples and logging them off the node. Yet these agents, particularly those doing frequent sampling, treat every sample as having the same value. In steady-state, this results in oversampling, whereas in highly-dynamic periods this can result in undersampling and the loss of important information. Online learning algorithms such as multi-armed bandits [32] can be used to smartly decide what telemetry to sample on the node or when to increase/decrease sampling while staying within the collection and logging budget.

Finally, watchdogs can benefit because failure conditions can be complex. Existing watchdogs check for simple conditions that are highly likely to indicate problems. This often means that they cannot detect problems until the problems are already affecting customer QoS. Using on-node learning offers the opportunity to detect problems and take mitigating actions earlier, as well as to detect and diagnose more complex problems directly on the node.

Summary, implications, and challenges. Production platforms run numerous agents of different classes on each server node. These

on-node agents are resource-constrained and may be delayed by other activities. We argue that resource control, monitoring/logging, and watchdog agents could benefit substantially from on-node learning. Agents from these classes represent 35% of the total node agents. They run frequently and perform crucial or costly operations where ML can lead to significant savings (*e.g.,* improved resource utilization). *The key challenge is producing safe, robust, and effective ML-based agents under the constraints of real cloud platforms.* Today, there are no systems that can help agent designers address this challenge.

3 ON-NODE LEARNING

On-node ML enables agents to become more agile and make smarter decisions, while considering fine-grained workload and resource utilization dynamics. Table 2 presents a selection of recent applications of ML to on-node management tasks that outperform static policies. These agents have different goals, employ different types of ML models, and learn from different telemetry. Though these prior works demonstrate the benefits of incorporating ML logic into on-node agents, they neglect the impact of different failures conditions on agent performance and correctness, making them less practical to deploy into production.

3.1 When is on-node ML necessary?

Fresh workload-tuned models and predictions. There are many workload dynamics that are only predictable a short window into the future. For example, the SmartHarvest and the SmartOverclock agents rely on extremely short-term signals to predict future CPU utilization (25ms into the future) and performance improvement from overclocking (1s into the future), respectively. For these use-cases, training on telemetry periodically logged to a centralized store would result in perpetually stale models for the current workload. Similarly, requesting predictions from centralized models to change a resource allocation could result in perpetually predicting workload behavior that has already happened. Instead, the models must be trained and served online on the node, and continuously updated to learn the latest behavior. This enables the agent to better respond to current workload behavior on the node.

Fine-grained telemetry. For many of these use cases, the opportunities for improved efficiency come from learning high frequency workload dynamics. In these cases, the models must learn from telemetry sampled at a high enough rate to capture these high-frequency effects. For example, the SmartHarvest agent captures CPU telemetry every 50 μs (the agent dedicates an otherwise idle core for capturing this telemetry; when there are no idle cores, there is nothing to harvest so the agent does not run), the SmartOverclock agent reads CPU counters every 100ms, and the SmartMemory agent samples page access bits up to every 300ms. This fine-grained telemetry cannot leave the node, as the size of the per-node data would likely cause performance issues for customers. For example, a single 16 GB VM whose memory is scanned every 300ms produces 100 MB of telemetry a minute.

3.2 On-node ML challenges and requirements

Bad input data. On-node learning agents collect telemetry to update the model and make decisions. At the scale of a cloud platform operating millions of nodes, telemetry collection can fail in a variety of ways (*e.g.,* misconfigured drivers, changes in data semantics between architecture or OS).

ML models are developed with implicit and explicit assumptions about data semantics, but will often continue to learn and produce predictions on data that violates these assumptions. The result is useless models trained on noise whose predictions should not be trusted. Instead, data assumptions should be specified and explicitly checked. In case of transient errors, if the invalid data can be detected and discarded, the model can still learn and provide useful predictions. Otherwise, learning on even small amounts of bad data can corrupt the model.

Poor model accuracy. There are many reasons why a model may have poor accuracy. For example, it may be learning from bad data (not all invalid data is detectable a priori). Or it may be trying to learn the behavior of a workload that violates some modeling assumption and is therefore unlearnable by this model (*e.g.,* randomly changing workload dynamics). Inaccurate models cause agents to take consistently bad actions, which can lead to impact to customer workloads. Instead, models must be evaluated continuously to ensure they meet accuracy expectations, and their predictions should not be used during periods of poor accuracy.

Unpredictable resource availability. Agents are not guaranteed any computational resources. During periods of high CPU demand on the host or expensive dataplane operations (*e.g.,* large amounts of virtual IO), agents will be throttled for arbitrary periods of time. Compute-intensive agents running close to their CPU limits can experience slowdowns resulting in stale models and predictions. As a result, the resource allocation or monitoring decisions made by an agent may be in effect for too long, or the agent may make decisions based on stale data. If not detected and handled appropriately, these delays can lead to unsafe agent behaviors (*e.g.,* negative impact on QoS) by taking actions after workload dynamics have shifted.

Node performance and reliability. Finally, all agents must ensure that they are not negatively impacting node performance or health in the face of opaque VMs. Not all data or learning issues can be prevented, and other environmental factors outside the scope of the agent may interfere with its operation (*e.g.,* VM live migration). Sometimes servers run in stressed or constrained modes, such as being oversubscribed or power-capped. In such settings, there can be little room for efficiency improvement from on-node learning and attempting to do so can harm customer or node health. As a last line of defense, agents must estimate their impact on client workloads and node health, and disable themselves if necessary.

4 SOL INTERFACE AND DESIGN

We design SOL to implement on-node ML agents that are safe to deploy alongside customer workloads by ensuring that they detect and mitigate all of the failure conditions from §3.2. SOL's API guides agent developers through the agent-specific logic needed to manage these conditions, while remaining highly extensible to different use cases. To implement a new agent in SOL, all developers have to do is (1) write functions to instantiate the API's function signatures, and (2) instantiate parameters for how often the functions need to run. The SOL runtime takes as input the functions and parameters, and manages scheduling and execution.

```
interface Model<D,P>
{
  D CollectData();
  void UpdateModel();
  Prediction<P> ModelPredict();

  bool ValidateData(D data);
  void CommitData(Time time, D data);
  Prediction<P> DefaultPredict();
  bool AssessModel();
}
```

Listing 1: `Model` interface. It is parameterized by the type `D` of the data collected and the type `P` of the prediction. Developers provide functions to instantiate these signatures.

Next, we describe the interface SOL exposes to agent developers, then discuss its runtime design and operation.

4.1 SOL interface

SOL is a lightweight C++ framework that exposes an API to agent developers which reflects the shared structure and failure modes of learning agents. The API is split into two groups of functions: `Model` and `Actuator`, each with their own sets of safeguards. The `Model` is responsible for providing fresh and accurate predictions on a best-effort basis. The `Actuator` makes control decisions at regular intervals (anywhere from milliseconds to minutes, depending on the agent), using predictions from the `Model` when available. The `Model` and `Actuator` run independently in separately scheduled loops so the `Actuator` can continue to operate safely and take regular actions when the `Model` is throttled or underperforming.

We adopt this split design to decouple the ML logic (`Model`) from the node management logic (`Actuator`), whether that is resource control, monitoring decisions, or watchdog failure detection. Enforcing strong abstraction boundaries simplifies agent design and ensures that the agent is designed from the ground up to operate safely even without predictions.

`Model` interface. The top part of the `Model` interface (Listing 1) specifies the three operations that all models take: (1) collect data to learn and predict on, (2) update the model with newly acquired data, and (3) use the model to make predictions. These three operations are called to form a single learning epoch. Often models need to collect several datapoints before learning and making predictions, so a single learning epoch can contain multiple data collection operations, but an epoch contains at most one model update and predict operation. The output of a successful learning epoch is a `Prediction` object that contains the predicted value and an explicit expiration time for the prediction. Data collection frequency, maximum duration, and the minimum and maximum number of data points that can be collected in a learning epoch are all configurable by the developer.

The rest of the `Model` interface is devoted to detecting problems and taking mitigating action when they occur. Every individual datapoint must be validated, and only if it successfully passes validation will it be committed to be used in the model. The data validation interface in SOL takes as input the most recently read data. It can be used to perform range checks or simple distributional checks with developer-defined data structures in the class implementation. There are limits to the extent individual telemetry data can be validated, but data validation helps ensure data points do not violate any testable properties of the inputs. In addition, developers must specify an `AssessModel` function that periodically checks whether the model accuracy or other relevant performance metrics are acceptable for the prediction task of the agent. While the model assessment is failing, SOL will intercept predictions before they can be passed to the `Actuator`. This means that the `Actuator` can assume that any predictions it receives are from a validated model.

```
interface Actuator<P>
{
  void TakeAction(Option<Prediction<P>> pred);
  bool AssessPerformance();
  void Mitigate();
  static void CleanUp();
}
```

Listing 2: `Actuator` interface. This interface is parameterized by the type P of the prediction. Developers provide functions to instantiate these signatures.

For some agents, there are useful fallback heuristics that can be used to make safe workload-aware decisions even without an accurate model. These safe heuristic decisions can be implemented in the `DefaultPrediction` function. SOL will send them to the `Actuator` instead when the model cannot produce an accurate prediction due to either data collection or model quality issues. *Default* predictions should allow the node to behave in a way that has minimal impact on the agent-specific safety metric (*e.g.,* customer QoS), at the possible cost of running at lower efficiency. However, even default predictions have an expiration time as they are still reliant on fresh telemetry and can become stale.

Default predictions can also be explicitly sent to the `Actuator` at any stage of the learning epoch. This short-circuits the current epoch and starts a new one. This is useful when the developer can detect an error ahead of time (*e.g.,* if a prediction is below a confidence threshold).

`Actuator` interface. By design, the `Actuator` interface (Listing 2) closely resembles the interface of an agent that does not use ML. It is a simple control function called `TakeAction` that is called either when new data becomes available or after a developer-specified maximum wait time has elapsed, whichever comes first. The only difference from non-learning agents is that learning agents use the prediction from a model to decide which action to take.

The `TakeAction` signature takes an `Option<Prediction>` type as an input. There may not always be a prediction available from the model (even a default prediction), by the time the `Actuator` must take an action, in which case the option type contains `None`. Even if there is a prediction available, it may already be expired if there were scheduling delays or throttling experienced by the agent. SOL detects scheduling delays by inserting various timestamp checks in the execution loop. It relies on the system clock for accurate timekeeping. When a fresh prediction is not available within the specified time frame for the `Actuator` to take an action, the agent should take a conservative, safe action to preserve customer QoS and node health, even if it comes at the cost of reduced efficiency.

The Actuator requires its own safeguard specified in functions AssessPerformance and Mitigate. AssessPerformance directly assesses the agent's behavior end-to-end, independently of the internal state of the model. Whenever it detects the safeguard-triggering condition, the Mitigate function is then called to allow the agent to take mitigating action. This safeguard serves as the last line of defense for the agent, and mirrors the existing approach in production, which requires node agents to have their own watchdogs. The Actuator safeguard should measure proxies for the safety metric of the particular agent and define acceptable impact to these metrics as justified by business needs. For example, a poorly performing SmartHarvest agent can starve customer workloads that need CPU resources. Hence, its AssessPerformance function monitors vCPU wait time for these customer workloads and triggers the safeguard when the wait time exceeds a certain threshold (as configured by the developer). The SOL runtime periodically evaluates AssessPerformance and calls Mitigate when the performance is unacceptable. The Mitigate function for SmartHarvest stops borrowing cores from customer VMs to stop the agent from impacting their performance. However, failing the Actuator performance check is often a lagging indicator of negative impact. The safeguards in the Model may detect and avoid problems before they trigger the Actuator safeguard, reducing the severity of impact.

Finally, the Actuator requires developers to provide an idempotent and stateless CleanUp function. This function can be safely called at any time (*e.g.,* by node SREs). It stops the agent and restores the node to a clean state, regardless of whether the agent is running normally, has crashed, or is hanging. SREs can also work with developers to define additional signals (*e.g.,* node health problems, frequent stalling of the agent) upon which agents should be cleanly terminated with the CleanUp function.

4.2 SOL runtime design and operation

Design principles. The key design decision in SOL is to decouple the potentially expensive data collection and ML component of the agent from the control component. Internally, SOL maintains two separate control loops running in separate threads. The Model control loop collects data, updates the model, and produces predictions to a message queue. The Actuator control loop consumes predictions from this queue when available and periodically takes a control action and monitors the end-to-end scenario performance.

The actuation logic is much simpler and less computationally expensive than the model logic, which may need to collect substantial amounts of telemetry and perform many mathematical operations to train the model or make predictions. The specific actuation varies (*e.g.,* collect monitoring data, making resource control decisions, trigger alerts), but the structure is the same. At the same time, as we discuss in §2, agents can run at best as soft real-time systems that may be throttled or delayed without warning.

By decoupling the expensive model logic from the lightweight actuation logic, we prevent the Model from starving the Actuator during these periods of heavy throttling. This provides an opportunity for the Actuator to take a safe action to prevent node or customer impact while the model may be completely unable to run.

Operation. Given an instantiation of the agent API, SOL automatically starts and runs the Model and Actuator control loops according to developer-provided schedules (Listing 3). The Model loop collects data at the frequency specified by the user until either enough data has been collected and validated or the maximum epoch time has elapsed. If enough data has been collected, SOL updates the model and makes a prediction. Otherwise, it short-circuits the learning epoch by sending a default prediction to the Actuator.

```
class Schedule
{
  // Model
  int data_per_epoch;
  duration data_collect_interval;
  duration max_epoch_time;
  duration assess_model_interval;
  // Actuator
  duration max_actuation_delay;
  duration assess_actuator_interval;
}
void main()
{
  Schedule schedule(config_file);
  Model* model = new OverclockModel();
  Actuator* act = new OverclockActuator();
  SOL::RunAgent(model, act, schedule);
}
```

Listing 3: Executing an agent. Once developers have implemented the SOL interface, they pass their implementation to the SOL runtime for scheduling and execution.

In addition, SOL assesses the model accuracy periodically (every K epochs as specified by the user). If the model fails the accuracy check, SOL continues to operate the Model control loop normally. However, SOL intercepts predictions and instead passes a default prediction to the Actuator. This still allows the model to be updated and produce predictions, hence providing the opportunity for the model to recover from a period of bad performance. At the same time, it prevents the Actuator from acting on bad predictions.

The Actuator waits on the prediction message queue for up to a maximum wait time. When new predictions are available, it immediately uses them to take actions. If a timeout occurs, SOL still calls TakeAction to provide an upper bound on the time between control actions in the agent. SOL also periodically checks the Actuator safeguard to detect behavior that could impact customer workloads or node health. If the safeguard is triggered, it halts the Actuator control loop until the unsafe behavior is no longer detected.

5 DEVELOPING AGENTS IN SOL

To build new agents in SOL, developers need to provide the implementation of the four common ML operations along with the various safeguards. SOL's APIs direct development efforts towards handling failure conditions ahead of time via the definition of safeguards. This requires developers to carefully reason through what conditions are appropriate to monitor and what mitigating actions should be taken in response. We argue this extra development burden up front is crucial, as it helps substantially reduce the complexity of managing learning agents in production. The operationalization complexity has been shown to contribute a significant part of the total cost for deploying ML in production [31].

We implemented three on-node learning agents using SOL. They differ in the type of node resources they manage, the input data and ML models they use, and the timescales they run at. Next, we discuss how these agents can benefit from learning, and their implementation in SOL. Unless otherwise stated, the various agent parameter values were selected based on experimental tuning. In §6, we demonstrate the consequences of running these agents unchecked during failures and how SOL minimizes these consequences by detecting and mitigating when failures occur.

5.1 CPU overclocking

CPU overclocking presents opportunities for substantial performance improvements on some workloads [17]. However, overclocking significantly increases power consumption and can shorten hardware lifetimes. As cloud platforms explore providing overclockable VM offerings, they want to balance the performance improvements with the extra power cost.

To address this problem, we created an intelligent on-node overclocking agent called SmartOverclock, which uses Q-learning [34], a simple form of Reinforcement Learning (RL). It monitors the average Instructions Per Second (IPS) performance counter across the cores of each VM and learns when to overclock the VM. At the end of every 1-second learning epoch, the agent uses the observed IPS and current core frequency to calculate the current RL state and reward. It then updates the RL policy and uses it to pick the frequency for the next learning epoch. Because the agent cannot directly observe workload-level metrics (*e.g.,* tail latency) inside opaque VMs, it assumes that a workload benefits from overclocking when higher CPU frequencies increase IPS. Though IPS is not a perfect proxy for identifying whether overclocking is beneficial, it works well for most optimized workloads. To balance exploitation of the policy learned so far with exploration of new frequencies, the agent uses the action selected by the RL policy 90% of the time and randomly picks a frequency 10% of the time.

Validating data. The agent collects multiple CPU counters and validates that they are within their expected ranges, discarding any data that fails this check, *e.g.,* the IPS value should be between 0 and `max_freq` $*$ `max_IPC`. Even a small fraction of bad data can cause the model to learn a sub-optimal policy and prevent workloads from benefitting from overclocking (see §6).

Assessing the model. A poorly performing RL policy can cause the agent to overclock workloads that do not benefit, resulting in wasted power. To detect a bad policy, the agent (in the `AssessModel` function) computes the difference, Δ_r, between the *observed* reward when overclocking and the *expected* reward from using the nominal frequency. It discards predictions if the average Δ_r over the last 10 epochs falls below a threshold. In this case, the agent continues to randomly explore, but overrides the RL-selected actions by always picking the nominal frequency as the default prediction.

Handling delayed predictions. Brief periods of wasted power are acceptable for the agent to recover from short scheduling delays. Thus, the Actuator will wait for up to 5 seconds (5 learning epochs) for a prediction. If it has not received an un-expired prediction at the end of this period, it takes the safe default action of setting the CPUs to the nominal frequency to avoid wasting power.

Safeguarding the Actuator. As the end-to-end safeguard for the Actuator, we define a factor α, using three CPU counters: α = (unhalted_cycles - stalled_cycles)/total_cycles. This factor serves as a binary indicator of whether a workload might benefit from overclocking. If α is low, the workload will not benefit much and overclocking would simply waste power. The Actuator monitors the 90^{th}-percentile (P90) of α values over the past 100 seconds and triggers the safeguard if this value is below a threshold. We use P90 to smooth transient drops in α, while quickly exiting the safeguard when activity increases again. The safeguard restores all cores to the nominal frequency in the `Mitigate` function.

Cleaning up. The `Cleanup` function kills any running SmartOverclock agents and then restores all cores to the nominal frequency.

5.2 CPU harvesting

The second agent we implement in SOL is the SmartHarvest agent from prior work [37]. We adopt the same model design and parameters values as used in [37]. This agent opportunistically "harvests" CPU cores that have been allocated to a (primary) set of VMs but are currently idle. It then loans the harvested cores to a special VM (called an ElasticVM), but must return the cores to the primary VMs as soon as they need them. Prior work has shown that ML is beneficial for this task [37], but it did not fully explore the design of safeguards to ensure safe and robust agent performance. We choose it as a case study to demonstrate the benefits of implementing previously explored ML use cases in SOL (see §6).

The agent uses a cost-sensitive classifier from the VowpalWabbit framework [3] to predict the maximum number of CPU cores needed by the primary VMs in the next 25 ms. It collects VM CPU usage data from the hypervisor every 50μs and computes distributional features over this data as input to the model.

Validating data. We perform range checks on the counter readings similar to those of the SmartOverclock agent. In addition, if the primary VMs use all their allocated cores during a learning epoch, it is impossible to distinguish whether they needed exactly that many cores, or whether they were under-provisioned during the epoch and the degree of that under-provisioning. Learning from this CPU telemetry can skew the model and cause it to systematically underpredict primary core usage. We therefore also discard any data collected during periods of full utilization by the primary VMs, as done in [37].

Assessing the model. The original SmartHarvest designers did not discuss approaches to assessing the accuracy of the learning model, instead relying on their version of the Actuator safeguard to detect and mitigate any problems. However, the Actuator safeguard, while important, is a lagging indicator of impacted performance. Assessing the model accuracy can detect some problems earlier. Thus, our implementation measures the percentage of time that predictions from the model lead to primary VMs running out of idle cores. If this percentage is high, the model safeguard is triggered.

Handling delayed predictions. Similar to SmartOverclock, our implementation of SmartHarvest sets a time limit on the wait time for a prediction produced by the model. The agent waits for a maximum of 100 ms (4 learning epochs) to account for its tighter harvesting control loop.

Safeguarding the Actuator. The agent uses a hypervisor counter reflecting how long virtual cores of a primary VM have waited for physical cores to run on as a proxy for workload QoS degradation. Long wait times indicate insufficient idle cores. The Actuator safeguard monitors the P99 wait time using the same approach as in [37]. If the value is high, it disables harvesting by giving all cores back to the primary VMs.
Cleaning up. `Cleanup` kills any running SmartHarvest agents then returns all harvested cores to the primary VMs.

5.3 Page classification for tiered systems

Our third SOL-based agent, called SmartMemory, targets managed two-tiered memory systems, where a slower and lower-cost byte-addressable memory (*e.g.,* persistent [19, 38] or disaggregated memory) sits behind the faster but expensive DRAM-based first tier. To efficiently use such systems, prior work exploited the highly-skewed popularity of pages in real-world workloads [19, 38]. Building on this idea, our agent seeks to identify pages as hot, warm, and cold, so that a small number of hot pages are stored in first-tier DRAM [19, 38], warm pages are on slow memory, and cold pages are compressed or not stored at all [21].

To determine page hotness, we can scan page access bits through the hypervisor [19, 29, 38]. Frequent scanning provides more fine-grained information about relative page access rates, but may also degrade workload performance from TLB misses. Each time a page's access bit is cleared, the page entry is flushed from the TLB.

Our agent uses ML to minimize the number of TLB flushes while still accurately classifying memory as hot/warm/cold. It uses Thompson Sampling [30, 35] with a Beta distribution prior, a well-known multi-armed bandit [32] algorithm that yields good performance in practice. The agent learns the best scanning frequency for each 2MB region of memory, divided into 512 4KB pages. The optimal scanning frequency is the lowest frequency that yields the same number of accesses as the maximum frequency. This forces hot batches to be sampled at the maximum frequency, while colder batches can be sampled much less often. In every epoch, the agent uses the Thompson Sampling models to decide how often to scan each batch, ranging from 300ms to 9.6s. At the end of each 38.4-second epoch (4x the maximum sampling period of 9.6s), the agent observes whether each batch was oversampled, undersampled (as approximated by number of consecutive access bits set), or well sampled, and updates the models accordingly. The model then uses the variable rate scans to estimate the minimal set of batches that contributed 80% of total memory accesses. It classifies these batches as hot, and the remainder as warm batches that are candidates for first-tier DRAM offloading. Similar to the heuristic used in previous work on cold memory detection [21], we treat batches that have been untouched for more than 3 minutes as cold and exclude them from scanning and our analysis.
Validating data. The access scanning driver will return an error code if it fails to scan or reset any access bits. In these cases, `ValidateData` fails the sample.
Assessing the models. The main risk from inaccurate models is that hot memory regions will be undersampled, leading the agent to conclude they are colder than they really are. The SmartMemory model randomly samples 10% of the batches at the maximum frequency and computes the total number of accesses to these batches. It uses this sample as ground truth to estimate the fraction of access bits missed by the model-recommended scanning rates. If the fraction of missed accesses rises above 25%, the model is deemed to be undersampling page accesses.

To provide safe default predictions under partial sampling or undersampling, the agent downsamples the access scans from all the batches to the lowest scanning frequency so that hit counts across different batches are directly comparable. It then targets a much more conservative 95% hottest batches to keep in first-tier DRAM, selecting only the coldest 5% of batches as candidates for warm memory using these downsampled hit counts. This helps protect workload QoS without completely disabling the second tier.
Handling stale predictions. Unlike our other agents, SmartMemory has no need to take any immediate mitigating action when predictions are delayed. It simply leaves the hot and warm pages where they are. If this decision becomes stale before the next prediction is received, the non-blocking system design triggers the Actuator safeguard to mitigate the problem.
Safeguarding the Actuator. The agent can directly observe the number of memory accesses to each tier using existing hardware counters. If the fraction of remote accesses over the last epoch is above the 20% target service level objective (SLO), the Actuator safeguard is triggered. In this case, the Actuator immediately migrates the 100 hottest batches in the second-tier memory back to the first tier. If the first tier does not have room for all 100 batches, it migrates as many as possible starting with the hottest batch.
Cleaning up. `Cleanup` kills any active SmartMemory agents and restores all second-tier batches back to the first tier until either all batches have been restored or the first tier is full.

6 EVALUATION

We evaluate (1) the utility of ML in each SOL agent we build, and (2) the efficacy of SOL's API in detecting failure conditions and mitigating their impact. Each agent we study manages a different resource. Therefore, the impact of failures on workload performance and node health also differs across agents. We begin by evaluating all safeguards using the SmartOverclock agent and include additional experiments for the SmartHarvest and SmartMemory safeguards where their behavior differs significantly from SmartOverclock.

6.1 Experimental Setup

Because agents running on each node are independent of each other, we run experiments on a single node and inject failures into the system to evaluate their resilience to these failures.

All experiments run on a two-socket Intel server with the Xeon Platimum 8171M processor capable of running at up to 2.6GHz, with 26 cores per socket and 384 GB DRAM. To reduce performance jitter for the customer VMs, we disable simultaneous multithreading, C-states, and Turbo-Boost. The server runs the Hyper-V hypervisor.

We run the agents in user-space on the root partition of Hyper-V. The overhead of running an agent is dependent on invocation frequency and computation overhead of various learning functions and safeguard checks. The SOL runtime manages the scheduling

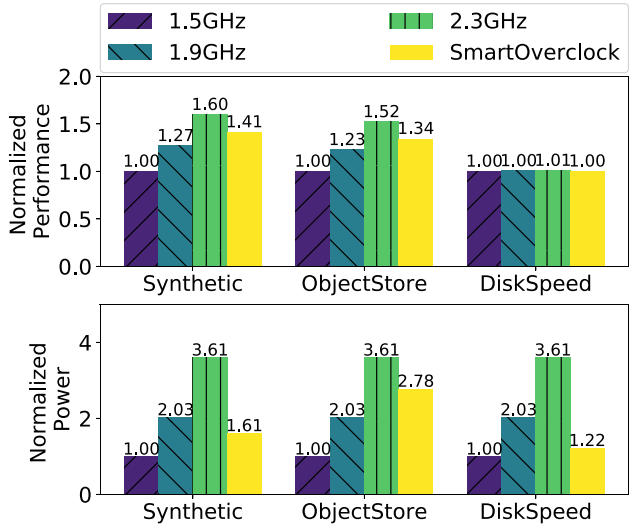

Figure 1: SmartOverclock learns to only overclock when the workload can benefit. Performance and power are normalized to the baseline values at 1.5GHz.

in user-space and runs in the same process as the agent, requiring very few resources.

6.2 SmartOverclock

We set the nominal server frequency to 1.5GHz and let the agent select from three possible CPU frequencies: 1.5, 1.9, and 2.3 GHz. Within an epoch, the SmartOverclock agent sets cores for a VM to the same frequency, but can change this frequency between epochs. While per-core frequency scaling is possible, the agent has no visibility into the thread scheduling (which governs per-core utilization) within the VM.

We first compare workload performance and power consumption of SmartOverclock to static policies that use a single frequency. The **Synthetic** workload simulates a server that periodically (every 100 secs) receives a batch of compute-intensive requests and processes them as quickly as possible, then is idle until the next batch arrives. This workload only benefits from overclocking during its request-processing phases. Performance is measured as the total time to complete a fixed number of batches. **ObjectStore** is a distributed key-value server running at high load that always benefits from overclocking. Performance is reported as P99 latency. **DiskSpeed** is a disk-bound workload that does not benefit from overclocking. Performance is reported as throughput in requests/sec.

Figure 1 shows the normalized peformance and power drawn by the three workloads at various static frequency settings and when using SmartOverclock. It shows that SmartOverclock provides the highest or second highest performance, indicating that it is overclocking workloads when they benefit. Statically overclocking the Synthetic workload at 2.3GHz only provides a 13% performance gain over SmartOverclock, yet uses twice as much power, demonstrating the inefficiency of static policies for dynamic cloud workloads. ObjectStore shows similar trends. DiskSpeed illustrates the case where SmartOverclock detects the workload's disk-bound behavior and keeps the frequency down, except for its intentional exploration of other frequencies.

SmartOverclock does not achieve the same performance as a static 2.3GHz frequency for CPU-bound workloads for two reasons: (1) agent exploration intentionally sacrifices short-term benefit for long-term adaptability, and (2) learning a model requires repeated observations to learn changes in workload dynamics. SmartOverclock sacrifices optimal peak performance for near-optimal performance and power usage on a wide range of workloads, achieving a higher performance/power ratio.

Invalid data. We now evaluate the impact of invalid data on SmartOverclock's model accuracy for the Synthetic workload. In Figure 2, we vary the percentage of bad data the agent collects by randomly returning out-of-range IPS readings to the agent a fixed percentage of the time. Without data validation, even 5% of invalid IPS readings causes a 17% drop in performance, while with data validation the workload still sees optimal performance. Eventually, too many invalid data readings will prevent the model from making a prediction at all and the scheduling delay safeguard will get triggered, returning cores to the nominal frequency.

Inaccurate model. We study the SmartOverclock model safeguard on all three workloads in Figure 3 by breaking the model, causing it to consistently select the highest frequency. Without the model safeguard, there is nothing to stop the agent from wasting power.On the DiskSpeed workload, this results in a 268% increase in power draw, whereas the model safeguard can detect this failure and increases total power draw by only 18%. ObjectStore benefits from overclocking and so a broken agent that always overclocks still achieves good results. However, the workload could change phases at any time without the agent changing its overclocking decision.

Delayed predictions. Next, we turn to the effectiveness of SOL's decoupled non-blocking design in preventing delayed predictions from impacting node health. We study the worst case occurence of a delay during phase changes in the Synthetic workload, which can cause the agent to waste power by overclocking an idle workload. We inject a 30-second delay in the Model thread when the workload finishes processing a batch and compare SOL's non-blocking Actuator to a blocking version that waits to change core frequency until a prediction is available. As Figure 4 shows, the blocking agent overclocks the workload for 30 seconds into its idle phase, increasing power consumption by 36%. The non-blocking agent waits a maximum of 5 seconds for a prediction from the model. In the absence of fresh predictions, it restores the node to a safe state (nominal frequency), consuming only an additional 3% of power.

Actuator safeguard. Fianlly, we evaluate the SmartOverclock actuator safeguard, which uses α to detect when the workload is in a stable phase of low CPU utilization. Many cloud workloads include VMs that are transiently idle for many minutes at a time (*e.g.*, a VM that runs periodic data processing jobs for 30 minutes every hour). During these idle periods, the Actuator safeguard completely disables overclocking to avoid wasting power. Figure 5 illustrates that the safeguard can detect and disable the agent during periods of low activity while remaining sensitive enough to quickly detect a period of higher CPU activity and re-enabling the agent.

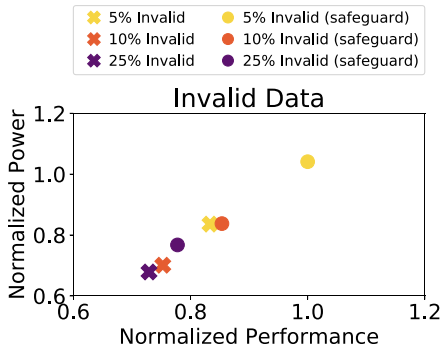

Figure 2: SmartOverclock data validation safeguard mitigates transient data errors. Power and performance are normalized to the ideal agent decision-making (all valid data).

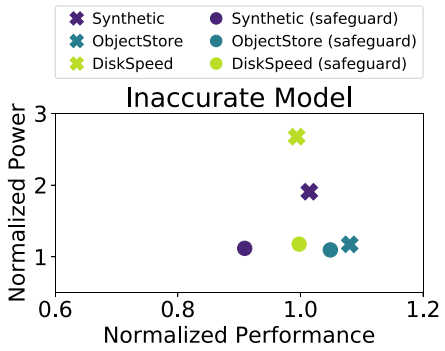

Figure 3: SmartOverclock model safeguard detects when RL overclocks without gains. Power and performance are normalized to the ideal agent decision-making (correct model).

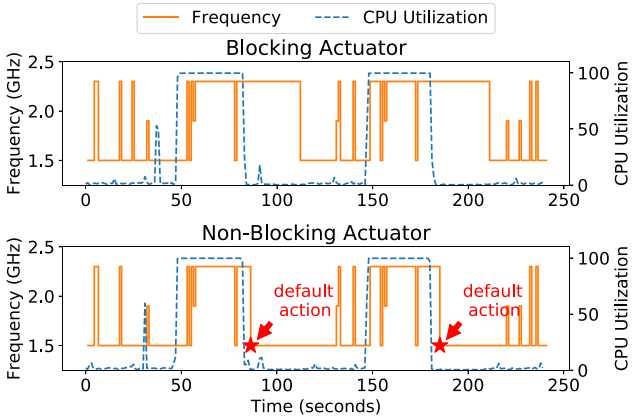

Figure 4: Non-blocking Actuator for SmartOverclock prevents wasted power when predictions are unavailable.

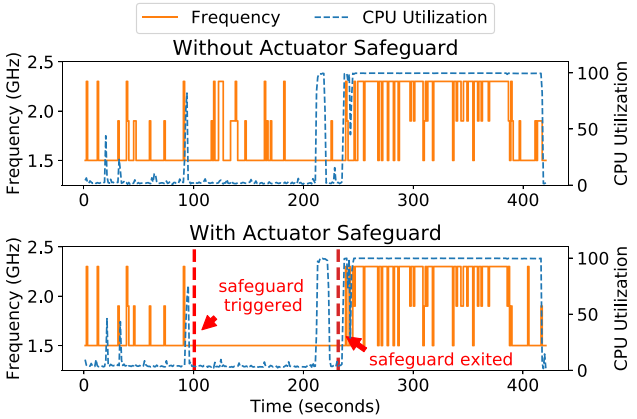

Figure 5: The SmartOverclock actuator safeguard reduces wasted power during long-lasting idle phases.

6.3 SmartHarvest

We next evaluate our implementation of the SmartHarvest agent in SOL. Prior work [37] provides a thorough evaluation of the benefits of machine learning for CPU harvesting, so we focus our evaluation on the additional safety provided by SOL's safeguards. When comparing the original implementation of SmartHarvest to the SOL implementation, we find that they have a similar number of lines of code – 1900 in the original compared to 1990 in SOL. However, the version in SOL contains the full set of safeguards required by the framework, while the original version lacks this functionality, making it more susceptible to operational issues. Guiding developers to ensure that their learning agents are appropriately hardened is a primary goal of SOL. Without this hardening, these issues can be difficult to detect and debug in production, hence reducing QoS and/or platform efficiency.

We evaluate the SmartHarvest agent when it tries to predict the CPU utilization of a co-located primary VM. We use either of two latency-sensitive workloads from TailBench [18] as the primary VM: **image-dnn** which performs image recognition and **moses** which does language translation. We measure performance of both workloads as their P99 latency.

Invalid Data. In the leftmost plot in Figure 6, we evaluate the SmartHarvest data validation safeguard, which discards observations when the primary VM is using all available cores. Without this safeguard, SmartHarvest consistently underpredicts the primary VM's CPU utilization in both workloads, causing the primary VM's P99 latency to increase by as much as 40%. With the safeguard, the impact on the primary VM's P99 latency is substantially less than 10% (the acceptable performance envelope in [37]).

Inaccurate model. In the case of a broken model (middle plot of Figure 6), the SmartHarvest model safeguard detects that the model is consistently underestimating the primary VM's CPU demand. When the safeguard is triggered, SOL switches to the default predictions which alleviate the impact on the primary VM's workload at the cost of harvesting fewer cores.

Delayed predictions. We see a similar impact on workload latency in the rightmost plot of Figure 6, when we insert 1-second scheduling delays during periods when the primary VM increases

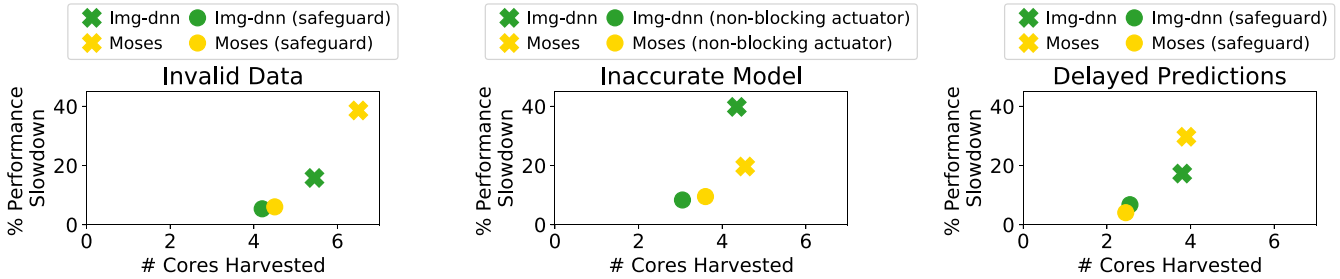

Figure 6: SmartHarvest safeguards: The left plot shows that the data validation safeguard prevents bad data from biasing the model to underestimate primary VM CPU demand, reducing the impact on customer workloads by up to 4x. The middle plot shows that the model safeguard reduces the impact of a broken model on workloads by up to 4x. The right plot shows that the non-blocking SOL implementation reduces the impact on workloads by up to 3x compared to a blocking agent.

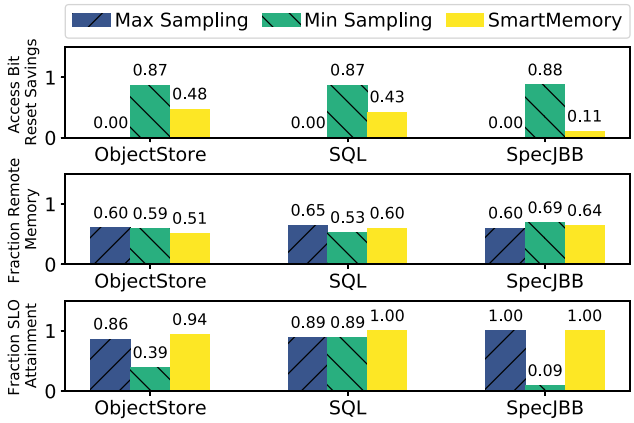

Figure 7: SmartMemory vs static access bit scanning.

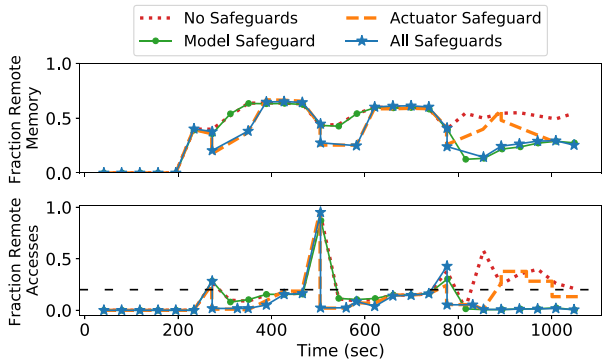

Figure 8: SmartMemory Model and Actuator safeguards.

CPU utilization. This worst-case scenario illustrates the importance of SOL's non-blocking design. If the agent blocks on a prediction from the model, a 1-second delay can cause up to a 30% increase in workload latency. This increase happens because during the delay, the primary VM's CPU utilization increases and it needs more cores, but the agent is blocked and cannot respond. The non-blocking agent has no information during the delay either, but it can quickly take the safe action of restoring all cores back to the primary VM.

6.4 SmartMemory

The SmartMemory agent handles delayed predictions and invalid data similarly to the other agents. Hence, we focus on (1) demonstrating the effectiveness of adaptive access bit scanning in reducing access bit resets and (2) the importance of SOL's safeguards in protecting workloads from too many slow tier-2 memory accesses.

In Figure 7, we compare the SmartMemory agent to two baselines without any safeguards: always scanning at the maximum frequency (300ms) and always scanning at the minimum frequency (9.6s). We evaluate on three workloads: **ObjectStore**, **SQL** (a standard OLTP benchmark executed on SQL Server), and **SpecJBB** (which executes SPECjbb2000 [1] for performance evaluation of server-side Java). For all workloads, the agent tries to maximize remote (tier-2) memory usage while ensuring that at least 80% of memory accesses are local as the service-level objective (SLO).

The top plot shows the reduction in access bit resets compared to the fastest frequency. SmartMemory reduces access bit scans by up to a 48%, while still reducing local memory size by 51% to 64% (middle plot). In the bottom plot, we observe the importance of SOL's safeguards. Access bit scans reflect only the current memory access patterns regardless of scanning frequency. Even when scanning at the maximum frequency, if the workload access patterns change, safeguards are needed to quickly mitigate SLO violations. Further, the bottom plot shows that sampling at the minimum frequency does not provide enough resolution to identify the hottest batches when targeting the 80% local accesses SLO, resulting in SLO attainment as low as 9%.

Model and Actuator safeguards. Figure 8 presents a more detailed evaluation of the SmartMemory Model and Actuator safeguards. We designed a workload that is difficult for SmartMemory to learn well: it oscillates between running SpecJBB for 150 seconds and sleeping for 80 seconds, resulting in frequent and rapid shifts in memory access patterns.

Without any safeguards, the SmartMemory agent only meets the SLO 66% of the time. When we add the Actuator safeguard, the agent can recover from instantaneous SLO violations immediately instead of waiting for the next learning epoch. We observe this effect at 250 seconds and 500 seconds, where the Actuator safeguard line on the bottom plot immediately drops back below 20% remote accesses. However, starting around 800 seconds, the models have consistently low accuracy for several epochs and we see that the Actuator safeguard takes multiple minutes to fully mitigate the SLO violation. In contrast, with the Model safeguard enabled, the agent is prevented from using the inaccurate predictions starting at 800

seconds, using the default predictions instead and avoiding the SLO violation in the first place.

Only SmartMemory with all safeguards enabled can both avoid using inaccurate predictions in the first place (Model safeguard), and quickly recover from SLO violations when they happen (Actuator safeguard). With all safeguards, SmartMemory meets the SLO 90% of the time, even on this intentionally difficult workload.

7 RELATED WORK

We are not aware of any prior characterizations of node agents in public cloud platforms or work on general and extensible frameworks for implementing safe and robust on-node learning agents.

Infrastructure for ML deployment. Centralized ML systems, where models are trained offline and served online, have become the standard deployment strategy [2, 5, 9–12, 14, 28]. Though useful for many scenarios, these systems cannot be used for the on-node learning tasks SOL addresses.

On-node ML. Recent works explored online learning for improving on-node resource efficiency or workload performance [4, 7, 8, 16, 25, 27]. Though effective for their particular use-cases, they did not propose general frameworks for implementing agents or address the deployment constraints of public cloud platforms (*e.g.,* the need to learn at the platform level from opaque VMs, instead of inside VMs or with application changes). SOL helps developers build agents that run safely outside of customer VMs without any visibility into or changes to them.

Safeguards for learning. There has been some exploration of the safety challenges involved in online ML [23, 24, 33, 37]. For example, the authors of [24] discuss a fallback policy when the model performs badly. SmartHarvest [37] focused on protecting the performance of customer workloads from poor predictions. None of these works helps developers with which issues to manage or how to build agents in a safe and robust manner.

8 CONCLUSION

This paper explored the challenges in improving production public cloud platforms by infusing online machine learning into their node agents. We first surveyed the existing (non-learning) agents in Azure and found that 35% of the 77 agents have the potential to benefit from learning. We then presented SOL, a general and extensible framework for developing on-node learning agents that can operate safely under various realistic issues, including bad data, scheduling delays, inaccurate models, and external interference. To demonstrate SOL, we implemented three agents using it and experimentally showed (1) the benefits of infusing learning into the agents, and (2) how the design of SOL ensures that they are robust to a variety of failure conditions.

ACKNOWLEDGEMENTS

We would like to thank our shepherd Akshitha Sriraman and the anonymous reviewers for many helpful comments. We would also like to thank John Thorpe for valuable preliminary investigations. This work was partially supported by the Stanford Platform Lab and its affiliates. Yawen Wang was also supported by a Microsoft Research Dissertation Grant.

REFERENCES

[1] 2006. SPEC JBB2000. https://spec.org/jbb2000/. Retrieved in January 2021.
[2] 2020. Torchserve. https://pytorch.org/serve/. Retrieved in January 2021.
[3] 2020. Vowpal Wabbit. https://github.com/VowpalWabbit/vowpal_wabbit/wiki. Retrieved in January 2021.
[4] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2018. Placeto: Efficient progressive device placement optimization. In *Machine Learning for Systems Workshop at the 32nd Conference on Neural Information Processing Systems.*
[5] 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. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* https://doi.org/10.1145/3097983.3098021
[6] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes, Girish Bablani, and Mark Russinovich. 2020. Toward ML-Centric Cloud Platforms. *Commun. ACM* 63, 2 (2020). https://doi.org/10.1145/3364684
[7] Vladimir Bychkovsky, Jim Cipar, Alvin Wen, Lili Hu, and Saurav Mohapatra. 2018. Spiral: Self-tuning services via real-time machine learning. *Blog post at https://engineering.fb.com/data-infrastructure/spiral-self-tuning-services-via-real-time-machine-learning* (2018).
[8] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. 2018. SmartChoices: Hybridizing Programming and Machine Learning. In *Reinforcement Learning for Real Life Workshop at the 36th International Conference on Machine Learning.*
[9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles.*
[10] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. 2015. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *Proceedings of the 17th Biennial Conference on Innovative Data Systems Research.*
[11] Dan Crankshaw and Joseph Gonzalez. 2018. Prediction-Serving Systems: What Happens When We Wish to Actually Deploy a Machine Learning Model to Production? *ACM Queue* 16, 1 (2018). https://doi.org/10.1145/3194653.3210557
[12] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation.*
[13] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013). https://doi.org/10.1145/2408776.2408794
[14] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems.* https://doi.org/10.1145/2541940.2541941
[15] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation.*
[16] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation.*
[17] Majid Jalili, Ioannis Manousakis, Íñigo Goiri, Pulkit Misra, Ashish Raniwala, Husam Alissa, Bharath Ramakrishnan, Phillip Tuma, Christian Belady, Marcus Fontoura, and Ricardo Bianchini. 2021. Cost-Efficient Overclocking in Immersion-Cooled Datacenters. In *Proceedings of the 48th Annual International Symposium on Computer Architecture.* https://doi.org/10.1109/ISCA52012.2021.00055
[18] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proceedings of the IEEE International Symposium on Workload Characterization.*
[19] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the USENIX Annual Technical Conference.*
[20] Alok Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. 2021. Prediction-Based Power Oversubscription in Cloud Platforms. In *Proceedings of the USENIX Annual Technical Conference.*
[21] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid,

et al. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3297858.3304053

[22] Anthony Liguori. 2017. Introducing the Nitro Hypervisor – the Evolution of Amazon EC2 Virtualization. https://www.youtube.com/watch?v=LabltEXk0VQ

[23] Kiwan Maeng and Brandon Lucia. 2020. Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3385998

[24] Hongzi Mao, Malte Schwarzkopf, Hao He, and Mohammad Alizadeh. 2019. Towards Safe Online Reinforcement Learning in Computer Systems. In *Proceedings of the 33rd Conference on Neural Information Processing Systems*.

[25] Nikita Mishra, John D Lafferty, and Henry Hoffmann. 2017. ESP: A Machine Learning Approach to Predicting Application Interference. In *Proceedings of the 14th IEEE International Conference on Autonomic Computing*. https://doi.org/10.1109/ICAC.2017.29

[26] Ravi Murty. 2019. Powering next-gen Amazon EC2: Deep dive into the Nitro system. https://www.youtube.com/watch?v=rUY-00yFlE4&t=2634s

[27] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture*. https://doi.org/10.1109/HPCA.2017.13

[28] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. In *ML Systems Workshop at the 31st Conference on Neural Information Processing Systems*.

[29] SeongJae Park, Yunjae Lee, and Heon Y Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*. https://doi.org/10.1145/3366626.3368125

[30] Daniel J. Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2018. A Tutorial on Thompson Sampling. *Found. Trends Mach. Learn.* 11, 1 (2018). https://doi.org/10.1561/2200000070

[31] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. In *Software Engineering for Machine Learning Workshop at the 28th Conference on Neural Information Processing Systems*.

[32] Aleksandrs Slivkins. 2019. Introduction to Multi-Armed Bandits. *Foundations and Trends® in Machine Learning* 12, 1-2 (2019). https://doi.org/10.1561/2200000068

[33] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2021. Automatically Enforcing Fresh and Consistent Inputs in Intermittent Systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3453483.3454081

[34] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*.

[35] William R. Thompson. 1933. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika* 25, 3/4 (1933).

[36] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*. https://doi.org/10.1145/2741948.2741964

[37] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the 16th European Conference on Computer Systems*. https://doi.org/10.1145/3447786.3456225

[38] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3297858.3304024