Project Final Report: Modeling and Generating Realistic Workloads

JianChen Zhao
jianchen.zhao@uwaterloo.ca

Abstract

As software systems become more complex and deployed on increasingly diverse platforms, the need for load testing and quality workloads is also increased. Load testing is required to ensure the proper functioning of a software system under a realistic environment and can find potential issues that functional tests cannot find. In order for load testing to be effective, one needs to input a realistic workload. Collecting user inputs is costly and complex. Thus, many research works propose automatic workload reproduction approaches. However, many of those approaches do not consider noise and the ability to extrapolate novel action sequences. We propose to use an interleaved HMM to learn the user behavior through event logs, which are then used to generate action sequences. We describe in detail the formulation of the approach and techniques used. We implement two variants of our approach, WorkLog_α and WorkLog_β. We show through an extensive evaluation that our approach can generate realistic yet diverse workloads while also being customizable to the use case.

1 Introduction

In today's dynamic and interconnected digital landscape, where user expectations for speed and reliability are higher than ever, workload testing is indispensable. For instance, Ticketmaster, a well-known ticket vendor, faced technical difficulties such as server crashes and increased latency during a pre-sale event for Taylor Swift's 2022 tour. Despite preregistrations for the event, their system faltered due to inadequate load testing [19]. This incident resulted in public backlash and multiple lawsuits. In contrast, during Prime Day 2023, Amazon handled an astounding number of requests, including 764 petabytes of EBS data transfer, over 15.35 trillion requests, 830 CloudTrail events, and 500 million CloudFront HTTP requests per minute. Amazon would not have achieved such a feat without rigorous load testing[2].

As software systems become more complex and are deployed on increasingly diverse platforms, assessing their robustness under varying workloads becomes paramount. Large-scale software systems like cloud services are deeply embedded into today's economy. Many businesses's critical operations rely on such software. To illustrate, the global end-user spending in public cloud services is more than half a trillion USD in 2023[8]. For operators of such large-scale software systems, a failure can result in significant financial loss and impact business worldwide[17]; Facebook, for example, lost an estimated 90 million USD to a 14-hour outage caused by misconfigured DNS[1]. Interestingly, such failures can not be attributed to functional bugs[26]. It is, therefore, crucial that systems are rigorously tested as a whole.

1.1 Background

Among the various system testing paradigms, load testing emerges as a critical aspect, focusing on the system's ability to handle different levels of expected user activity. This form of testing empirically guarantees that a software application can meet its performance objectives under realistic working conditions.

Load testing involves simulating different scenarios and conditions that mimic the actual usage patterns and demands the software may encounter in a production environment. Subjecting the system to *workloads*, including typical and peak loads, reduces system failure risks while helping identify potential bottlenecks, vulnerabilities, or performance degradation that may not arise in functional tests [24, 22, 9, 13, 6].

We define a workload as a sequence of actions to which we can subject a system under test. In research and practice, there are many ways one might obtain workloads.

The most basic form of load testing consists of replaying the system against a manually collected dataset of user actions. This approach requires extensive tooling of the deployed system to capture the user actions and ample storage to accommodate the large number of workloads necessary to capture the variance in user actions. Many prior works have explored the automatically recovering workloads to work around these issues.

However, automatically generating workloads is no trivial task. There are two main challenges. First, the generated workload must be realistic. The action patterns in the workload must be representative of actual field observations. This property is essential to the usefulness of load testing. Testing on unrealistic data cannot ensure that the system under test can handle actual user actions. Second, real user actions have high variance; that is, there is a diverse number of action patterns in production environments. Hence, automatic workload generation techniques must balance the variance and representativeness of the recovered workload.

1.2 Related Work

Prior works on workload generation have explored automatically recovering workloads from various readily available sources, which are resampled for replay.

Many proposed approaches abstract away user behaviors in workload and only consider specific metrics. This reduces the information needed to be stored for replay. [22, 9] use CPU usage metrics for high-level representation of user behaviors. Similarly, [12, 28, 4, 21] characterize the load by the I/O usage, while [11] looks at other system resource usage to model the workloads. While such coarse-grained approaches are efficient and easy to maintain, information about essential characteristics of the load might be lost.

As demonstrated in [9], more precise characterization of workloads may lead to better performance. [28] explored using a hidden Markov model to find patterns of user behaviors and found that only a small amount of data is required to construct an adequate model of the user behaviors.

To balance the granularity of the recovered workloads, [24, 25, 23, 27, 13] distinguishes different user behaviors by clustering user actions extracted from event logs or system traces. [5] adds to this approach by also considering sequences of user actions and their context, helping find more representative user behaviors without sacrificing the variance of the recovered workloads and introducing some adaptability to the approach. While this approach can achieve a good balance, it suffers from two problems. First, the approach clusters users based on their behaviors to sample actions and action sequences of a recovered workload using representative users. In effect, the diversity of the generated workload is restricted by the size of the source used to recover the initial workload. Second, event logs and traces are noisy. Actions are thus recovered by filtering on some user identifiers. However, those identifiers are not always present, and the recovered actions may be incomplete [29]. We extend [5] with ideas from [28] to achieve the ability to generate realistic yet novel and high-variance workloads.

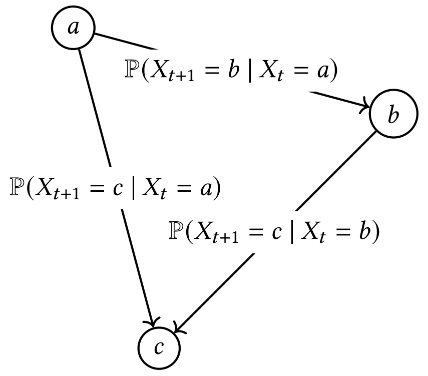
1.3 Contributions

To tackle the challenges above, we propose an extensible model of user behaviors found in event logs based on a mixture of Hidden Markov Models (HMM) to allow the dynamic generation of workloads unconstrained by the size of the initial recovered workload. We evaluate our model and demonstrate its performance and generalizability. We also propose directions for future improvements.

1.4 Organization

We present our model in detail in section 2 while highlighting challenges in training such a model. We show in section 3 how it can be integrated to generate workloads dynamically. We evaluate it in sections 4 and 5, and discuss tradeoffs and potential improvements in section 6. Finally, we conclude in section 7.

Figure 1. An example of a graph of the states of a process, with associated state transition probabilities. This graph is analogous to a control flow graph.


a
b
c
$\mathbb{P}(X_{t+1} = b \mid X_t = a)$
$\mathbb{P}(X_{t+1} = c \mid X_t = a)$
$\mathbb{P}(X_{t+1} = c \mid X_t = b)$

2 Modeling Event Logs

Event logs and system traces are discrete-time processes. Given time steps $1, 2, ..., t, ..., \tau - 1, \tau$, events y_t are observed according to an underlying process, that is, the control flow of a running software system.

2.1 Control Flow Graph

The control flow graph (CFG) of a program can be understood as the graphical representation of its state transitions at runtime. Nodes in this graph represent sequences of statements executed sequentially without branching except at the entry and exit statements. Edges represent the control flow and its branching between the basic blocks.

The CFG can also be seen as a description of the underlying process that produces outputs and, for our purposes, event logs and system traces. However, to get a complete picture of the production process, we need to compute the CFG for the whole program, or in other words, the interprocedural CFG (ICFG). Computing this ICFG is resource-intensive, impractical, and often outright impossible. It must thus be approximated[29].

2.2 Hidden Markov Chains (HMM)

Instead of directly approximating an ICFG, we can fit a hidden Markov model (HMM) on a program's output to learn its states and transitions at runtime. In this approach, we do not require the use of source code. The learned graph, where nodes are states and edges are transitions, is an indirect approximation of the ICFG with respect to the output. That is, it models the process that generates the outputs.

A HMM $\mu = (X, Y)$ consists of two stochastic processes: $X = \{X_t : 0 \leq t \leq \tau\}$ is a Markov process that is not directly observable, hence hidden, and $Y = \{Y_t : 0 \leq t \leq \tau\}$ that is directly observed. Each X_t only depends on the previous X_{t-1} and each Y_t only depends on X_t.

For simplicity, we assume that X is ergodic; that is, the probability of getting from every state to every other state is non-zero. This assumption should hold in practice; for load testing, we are concerned with system states that are reoccurring and don't lead to the termination of the system.

Figure 2 shows a visualization of a HMM. At each time step, the state of the process transitions to a new state. We denote the set of all states as $\mathcal{S}$. Along with every transition, an observation is emitted. We denote the set of all observations as $\mathcal{O}$. At a given discrete time t, suppose the hidden process is at a state $x \in \mathcal{S}$. The probability of transitioning to a state $x' \in \mathcal{S}$ at time $t + 1$ is given by

$$a_{x,x'} = \mathbb{P}(X_{t+1} = x' \mid X_t = x). \tag{1}$$

The values of a are the *transition probabilities.* At time $t = 0$, the probability of being in a state x is given by the prior distribution with

$$\pi_x = \mathbb{P}(X_0 = x). \tag{2}$$

The values of pi are the *prior state probabilities.* The probability of emitting, or equivalently, observing, the event y at time t when the state is x is

$$b_{x,y} = \mathbb{P}(Y_t = y \mid X_t = x). \tag{3}$$

The values of b are the *emission probabilities.*

A HMM intuitively models event logs and system traces. The process Y is the observed events, while the process X approximates the interprocedural control flow. The state transitions represent the branching in the control flow between statements that produce events. Because this abstraction does not consider the actual user inputs to the software system, the branching behavior of the program is captured by the transition probabilities $a_{x,x'}$.

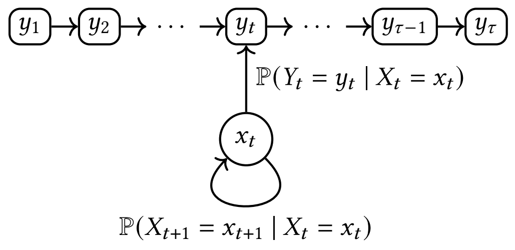
Since, in a control flow of a program, a single statement should only be able to emit at most one type of event, it may be practical to set $\mathcal{S} = \mathcal{O}$ and $b_{i,k} = \delta(i, k)$ where δ is the indicator function. However, we note that an ICFG is hard to approximate. Thus, it is more desirable to keep the set of hidden states $\mathcal{S}$ flexible, as the set of observations is fixed to the actual observations in event logs and system traces.

2.3 Interleaved HMM

While a single HMM can intuitively model a single process, A software system often comprises multiple processes. For example, a single system might run multiple programs, each utilizing multiple threads. Furthermore, event logs from each process may be collected and gathered in a centralized storage.

Individual threads can also communicate with each other, possibly across different hardware. Thus, data flows across thread boundaries. In this case, processes may not be restricted to a single thread. Instead, it is more suitable to let

Figure 2. A diagram representing a HMM. $y_1, y_2, ..., y_\tau$ are observations; they are emitted with probability $\mathbb{P}(Y_t = y_t \mid X_t = x_t)$. x_t is the hidden state at time t, where the next state is transitioned to with probability $\mathbb{P}(X_{t+1} = x_{t+1} \mid X_t = x_t)$.


y_1
y_2
$\cdots$
y_t
$\cdots$
$y_{\tau-1}$
y_τ
$\mathbb{P}(Y_t = y_t \mid X_t = x_t)$
x_t
$\mathbb{P}(X_{t+1} = x_{t+1} \mid X_t = x_t)$

the hidden states model the states of the data flow of the whole system.

Following this parallel computation paradigm, we can mix multiple HMMs. This mixture of HMMs can be encoded into a single HMM, called *interleaved HMM*[18]. All constituent hidden Markov chains produce observation events in the same sequence, but at any time step, only one individual chain can transition as dictated by some random process C.

Consider the random variable X_t for a state at type t. It can be defined as composition of states of m different HMMs μ_k for $1 \leq k \leq m$:

$$X_t = (S_{\mu_1,t}, S_{\mu_2,t}, ..., S_{\mu_m,t}, C_t). \tag{4}$$

C_t is a random variable used to index which chain is allowed to transition. It can be defined in different ways, including as an observation of another additional HMM, but for simplicity and following the fact that interleaving in even logs is mostly random, we choose that

$$\alpha_c = \mathbb{P}(C_t = c). \tag{5}$$

where α_c is a scalar probability representing the rate at which a chain is chosen. Intuitively, it is the probability that the underlying process of the constituent chain is chosen to generate an event. The new states of the interleaved HMM encode the state of all the constituent HMM using indexing tricks, as shown in [18].

At any time t, the probability of the constituent chain μ to transition from state s to s' is then

$$\mathbb{P}(S_{\mu,t+1} = s' \mid S_{\mu,t} = s, C_t = c_t) = \begin{cases} a_{\mu,s,s'} & \text{if } c_t = \mu, \\ \delta(s, s') & \text{otherwise,} \end{cases} \tag{6}$$

where δ is the indicator function. Then, eqs. (1) to (3) are redefined as follows:

$$a_{x_{t+1},x_t} = \alpha_{c_t} \prod_{\mu} \mathbb{P}(S_{\mu,t+1} = s_{\mu,t+1} \mid S_{\mu,t} = s_{\mu,t}, C_t = c_t), \tag{7}$$

$$\pi_x = \alpha_c \prod_{\mu} \pi_{\mu,s_\mu}, \tag{8}$$

and

$$b_{x,y} = b_{c,s_c,y} \tag{9}$$

Equations (4) and (7) indicates an explosion of states. Indeed, we discuss in detail how this impacts the model's training and possible remedies in section 3. Nonetheless, we will see in section 5 that a smaller model might be more desirable in line with the findings of [9].

3 Approach

We propose an automatic workload generation approach that leverages generative models and their unsupervised learning capabilities. Similar to [28], we use a mixture of HMM. We use the interleaved HMM described in section 2.3. We first extract sequences of actions that will constitute a workload. We use event logs as a proxy, as they are widely available, capture user behavior, and are relatively less complex to maintain than directly captured user inputs.

In this approach, only a small set of user actions must be collected to train the model. Once fitted, the model can extrapolate realistic synthetic workloads with high entropy.

Unlike [5], we do not consider clusters of users. As event logs are noisy and does not contain enough user identifiers [29] to relate actions to users. Instead, we rely on the fact that the interleaved HMM will eventually learn distinct patterns and behaviors in different constituent chains.

3.1 Collection Action Sequences

The first step in our approach is collecting action sequences. Traditionally, actions are obtained by collecting user inputs, after which they are organized into workflows. However, as pointed out in [5], this method is not cost-efficient and often times impractical due to the size and heterogeneity of large software systems.

Following the approach of [5], we use event logs as a proxy to user inputs to mirror the user behaviors. Logs also contain system behavior information that is not present in user inputs. This way, we can model the user behaviors and their interaction with different subsystems.

One caveat is that collecting actions from logs requires manual intervention. Once actions have been identified in logs, they need to be manually implemented to be replayed for load testing.

3.2 Fitting a HMM

Once we have a dataset of action sequences, it is used to train an interleaved HMM. In the context of Markov chains, actions are observed events. The traditional algorithm still applies to interleaved HMMs since they are merely regular HMMs encoding the states and probabilities of multiple constituent chains.

Consider a HMM μ with state sequence $\langle X_1, X_2, ..., X_\tau \rangle$ and an event sequence $\langle Y_1, Y_2, ..., Y_\tau \rangle$. To fit this HMM on a hidden process, we need to find parameters a, b and π of mu, by maximizing the likelihood $\mathcal{L}(\mu|\langle Y_t = y_t \rangle_{t=1}^{\tau})$. For some observed events $\langle y_1, y_2, ..., y_\tau \rangle$, it can be written as

$$\begin{aligned}\mathcal{L}(\mu|\langle Y_t = y_t \rangle_{t=1}^{\tau}) &= \mathbb{P}(\langle Y_t = y_t \rangle_{t=1}^{\tau}). \\ &= \sum_{x_\tau} \mathbb{P}\big(\langle Y_t = y_t \rangle_{t=1}^{\tau}\,,\; X_\tau = x_\tau\big)\end{aligned} \tag{10}$$

Let $p_\mu(\langle y_t \rangle_{t=1}^{\tau}\,,\; x_\tau) = \mathbb{P}\big(\langle Y_t = y_t \rangle_{t=1}^{\tau}\,,\; X_\tau = x_\tau\big)$. By recursively applying the chain rule that for $1 \leq t \leq \tau$

$$\begin{aligned}&p_\mu(\langle y_t \rangle_{t=1}^{\tau}\,,\; x_\tau) = \\ &\quad b_{x_\tau, y_\tau} \sum_{x_{\tau-1}} a_{x_\tau, x_{\tau-1}} p_\mu(\langle y_t \rangle_{t=0}^{\tau-1}\,,\; x_{\tau-1}).\end{aligned} \tag{11}$$

and with the base case

$$p_\mu(\langle\rangle\,,\; x_0) = \mathbb{P}(X_0 = x_0). \tag{12}$$

Using eqs. (10) to (12) we can construct algorithm 1, the forward algorithm, to compute the likelihood. First, we initialize for all states x a prior probability p_x. Second, we iteratively update p_x using eq. (11). Finally, the likelihood is obtained by taking the sum for all states according to eq. (10).

Algorithm 1 The forward algorithm. It computes the likelihood using eqs. (10) to (12).

```
1: procedure Forward($\langle y_t \rangle_{t=1}^{\tau}$)
2:     for $x \leftarrow \mathcal{S}$ do
3:         $p_x \leftarrow \pi_i$
4:     for $t \leftarrow \{1, 2, ..., \tau\}$ do
5:         for $x' \leftarrow \mathcal{S}$ do
6:             $p_{x'} \leftarrow b_{x', y_t} \sum_{x \in \mathcal{S}} p_x a_{x, x'}$
7:     return $\sum_{x \in \mathcal{S}} p_x$
```

Then, the fitted model $\hat{\mu}$ is one that maximizes the likehood:

$$\hat{\mu} = \arg\max_{\mu} \mathcal{L}(\mu \mid \langle Y_t = y_t \rangle_{t=1}^{\tau}). \tag{13}$$

Since the forward algorithm's gradient can be automatically computed by auto differentiation frameworks such as Jax[3], eq. (13) can be solved using stochastic gradient descent algorithms instead of the traditional expectation maximization algorithm[18]. Specifically, I used the Adam[15] optimizer on a synthetic dataset described in section 4.3. I implemented the forward algorithm in log space for better numerical stability, and thus, the loss to be minimized is the negative log-likelihood:

$$l = -\log \mathcal{L}(\mu \mid \langle Y_t = y_t \rangle_{t=1}^{\tau}). \tag{14}$$

The forward algorithm has space complexity $O(n)$ and time complexity $O(nT)$ where n is the number of possible states and T is the length of the input sequence. In the context of interleaved HMM μ, there are K individual HMMs μ_k, the state of μ must encode the states of all μ_k. Thus, no matter the encoding, the time and space complexity of the forward algorithm will be exponential with respect to K. Training the interleaved HMM is an intractable problem[16]. However, we find in section 5 that a small model is enough to generate adequate workloads.

3.3 Generating Workloads

Once the model has been trained, we can use it to generate workloads one action at a time. There are many ways we can generate. One way is to iteratively select single constituent chains and apply algorithm 2 on each to produce separated action sequences. However, the method we adopt instead is to apply algorithm 2 directly to the interleaved HMM to obtain a sequence of interleaved actions as the whole workflow. Intuitively, this is closer to how multiple user inputs interact in real workloads.

Algorithm 2 takes an initial state x and a desired sequence length τ. It iteratively updates x by calling a random choice function with weights given by the transition probabilities a. This corresponds to the transition step. After each transition, an action is chosen to be emitted using the same random choice function, this time with weights given by emission probabilities b.

Algorithm 2 Generates a sequence of actions

```
1: procedure GENERATE($x, \tau$)
2:     for $t \in \{1, 2, ..., \tau\}$ do
3:         $x \leftarrow$ CHOOSE($\mathcal{S}, a_x$)
4:         $y_t \leftarrow$ CHOOSE($O, b_x$)
5:     return $\langle y_1, y_2, ..., y_\tau \rangle$
```

4 Experimental Setup

We have implemented the interleaved HMM using the Flax[14] framework based on Jax[3]. To achieve better numerical stability, algorithms 1 and 2 are implemented in log space. We evaluate two variants of our approach.

With the first variant, which we call WorkLog$_\alpha$, we train an interleaved HMM directly using algorithm 1. The number of subchains and states it are both limited to 4 to reduce computation overhead.

Alternatively, with the second variant, WorkLog$_\beta$, we train each constituent chains separately using algorithm 1. Although computationally more efficient, it is the worst approximation.

4.1 Research Questions

RQ 1 *How does our approach perform for real workloads?* Can we say that our approach can produce realistic workloads? In practice, while HMMs can naturally model stochastic processes, the model might not be accurate due to the limitation in the number of states. Thus, we investigate the realism versus variance tradeoff.

RQ 2 *How does the approach perform under different settings?* Due to the many tunable parameters and workload characteristics, that is, the number of constituent chains in the interleaved HMM, the number of states in each chain, the number of actions, and the training batch sizes, it is important to know which configuration yields better results.

4.2 Evaluation Metrics

We use two different metrics to evaluate the realism and randomness of each variant of our approach.

4.2.1 Throughput Delta.

The first metric is Cliff's delta[10, 7] of the *throughput* of the actual action sequence against that of generated action sequences. We define throughput as

$$\mathbb{P}(\langle Y_t = y \rangle_{t=1}^{\tau}) \tag{15}$$

for a specific action y. Following suggestions from [20], we use the following interpretation of Cliff's delta: ≈ 0.01 indicates a very small difference, ≈ 0.2 is small, ≈ 0.5 is medium, ≈ 0.8 is large, ≈ 1.2 is very large and ≈ 2 is huge.

4.2.2 Perplexity.

Using the negative log-likelihood as a cross-entropy, we define *perplexity* as

$$\mathbb{P}(\langle Y_t = y_t \rangle_{t=0}^{\tau})^{-\frac{1}{\tau}} \tag{16}$$

for a sequence of observation $\langle y_t \rangle_{t=0}^{\tau}$. Because the interpretation of this metric is relative to the model, we only use this metric to compare a model against variants of itself. Usually, for example, with language models, perplexity can be interpreted as the model's randomness and confusion. However, we adapt this metric to measure the randomness of the generated workloads. Thus, higher is better.

4.3 Datasets

We use both datasets from real software projects and synthetically generated datasets.

4.3.1 Real Datasets.

To answer RQ 1, we use datasets taken from [5]. They consist of data from two open-source projects, Apache James and OpenMRS, and one industrial project, Google Borg.

Apache James is an enterprise mail server. [5] used JMeter to generate a workload of around 2000 users and captured its log. Only two types of actions are considered: "SEND," where a user sends an email, and "RECEIVE," where a user receives and loads an email.

OpenMRS is a web medical record service. [5] used the default demo database with over 5000 patients and 476,000 user actions. Four action types are considered: "ADD", "DELETE", "SEARCH" and "EDIT". They used JMeter and injected more randomness to simulate a more realistic workload.

4.3.2 Synthetic Datasets.

To answer RQ 2, we use a synthetic dataset, which we generate using a hardcoded interleaved HMM. The parameters of the HMM used to generate the dataset are not shared with the HMM under evaluation. Since the model is stochastic, it will not learn the same parameters as the HMM used to generate the dataset. By

Table 1. Throughput delta on the Apache James dataset.

Apache James			
	throughput	WorkLog$_\alpha$	WorkLog$_\beta$
SEND	2.746	**0.0322**	0.0835
RECEIVE	7.254	**0.0322**	0.0835
Mean	5.000	**0.0322**	0.0835

Table 2. Throughput delta on the OpenMRS dataset.

OpenMRS			
	throughput	WorkLog$_\alpha$	WorkLog$_\beta$
ADD	2.848	**0.0163**	0.0631
DELETE	3.080	0.0285	**0.0151**
EXIT	2.066	0.0395	**0.0305**
SEARCH	2.006	**0.0035**	0.0374
Mean	2.500	**0.0220**	0.0365

Table 3. Throughput delta on the Google Borg dataset.

Google Borg			
	throughput	WorkLog$_\alpha$	WorkLog$_\beta$
SUBMIT	3.453	0.0447	**0.0423**
SCHEDULE	1.009	**0.0529**	0.2128
FAIL	0.398	**0.0978**	0.2230
FINISH	3.472	**0.0984**	0.1286
EVICT	1.636	**0.0459**	0.1282
KILL	0.032	0.0060	**0.0010**
Mean	1.667	**0.0576**	0.1228

varying the parameters of the hardcoded HMM, we can obtain synthetic datasets with different characteristics, which we leverage to reveal insights about the variants of our approach.

In our experimentation, we use a small sample of approximately 1000 actions for each dataset. 90% of each dataset is used to train the model, while 10% is used for evaluation. That is, our model is evaluated on unseen data.

5 Results

RQ 1: How does our approach perform for real workloads?

To answer this question, we evaluated WorkLog$_\alpha$ and WorkLog$_\beta$ against Apache James, OpenMRS, and Google Borg datasets. For each dataset, we train a new interleaved HMM, each configured with the correct number of action types. Then, we generate a matching amount of sequences. Finally, for each action type, we compute the Cliff's delta of the throughput of sequences from the original dataset against the throughput of generated sequences.

Table 4. Performance of WorkLog$_\alpha$ and WorkLog$_\beta$ on synthetic datasets. "Delta" is the Cliff's delta of the throughput.

	WorkLog$_\alpha$		WorkLog$_\beta$	
	delta	perplexity	delta	perplexity
Easy	**0.0304**	3.6973	0.0601	**14.916**
Wide	**0.0311**	3.9588	0.1025	**15.919**
Hard	0.7370	4.1623	**0.7364**	**16.123**

We present the results in tables 1 to 3. Overall, WorkLog$_\alpha$ seems to perform better than WorkLog$_\beta$. This is expected due to the fact that WorkLog$_\beta$ uses an approximation of algorithm 1 during training. However, we note that WorkLog$_\beta$ is an order of magnitude faster to train.

Both WorkLog$_\alpha$ and WorkLog$_\beta$ perform well on the Apache James dataset, with very small deltas and with WorkLog$_\alpha$ performing slightly better than WorkLog$_\beta$. Coincidentally, the models have a constant delta for both types of actions, potentially indicating that both models learned as many user behavior patterns as their randomness would allow.

On the OpenMRS dataset, the difference in throughput delta between the two variants starts to blur. For the "ADD" and "SEARCH" action types, WorkLog$_\alpha$ performs slightly better, whereas for the "DELETE" and "EXIT" action types, WorkLog$_\beta$ is better.

This is because users behave differently for different actions, and WorkLog$_\beta$ learns some patterns faster. In table 3 is where we see WorkLog$_\beta$ starts to struggle. WorkLog$_\alpha$ performs better for all action types except for "SUBMIT" and "KILL". On this dataset, WorkLog$_\beta$ has a larger but still small delta.

Finding 1: Overall, both variants can reproduce realistic workloads. WorkLog$_\alpha$ performs slightly better on larger datasets.

RQ 2: How does our approach perform under different settings?

To answer this question, we first evaluate both WorkLog$_\alpha$ and WorkLog$_\beta$ on three different configurations of our synthetic dataset. We train a new model for each dataset instance and measure both the throughput delta and the perplexity.

The easy dataset is generated by an interleaved HMM with 4 constituent chains of 4 states each. The wide dataset is generated by an interleaved HMM with 16 constituent chains of 4 states each. The hard dataset is generated by an interleaved HMM with 4 constituent chains of 16 states each.

The results are recorded in table 4. We can observe that both variants produce sequences that are more realistic, e.g., have a smaller throughput delta, on datasets with a smaller number of underlying states. They struggle to reproduce the hard dataset, which has a higher number of underlying

Figure 3. Performance of WorkLog$_\beta$ versus number of constituent chains.

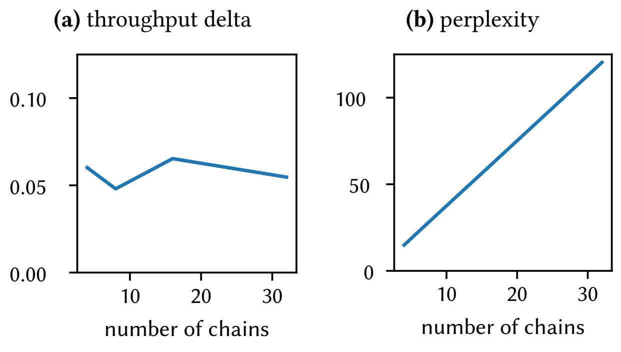
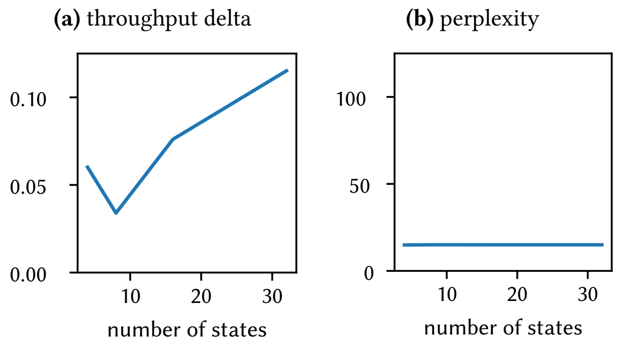


Figure 4. Performance of WorkLog$_\beta$ versus number of states of each constituent chain.



states. The number of constituent chains seems to not have an influence on the performance of WorkLog$_\alpha$, as indicated by the wide dataset. We also note that, overall, WorkLog$_\beta$ has higher perplexity, indicating more randomness.

Finding 2: Both variants struggle when the number of underlying states are increased. WorkLog$_\beta$ can produce more random sequences.

Since WorkLog$_\beta$ is more flexible regarding tunable parameters, we evaluate the influence of each. We investigate the effect of changing the number of constituent chains shown in fig. 3 and the number of states in each constituent chain shown in fig. 4.

We can observe that varying the number of constituent chains does not significantly affect the throughput delta. However, perplexity increases. Conversely, we observe that as the number of states increases, the throughput delta also increases, but the perplexity stays constant. This confirms [9]'s findings that using a more precise user behavior model is better.

Finding 3: Increasing the number of constituent chains increases perplexity and increasing the number of states increases throughput delta of WorkLog$_\beta$. We recommend using more chains with less states.

6 Discussion

From our evaluation results presented in section 5, we can conclude that our approach can produce realistic workloads while also adding significant bits of randomness.

We also note that our approach is highly customizable in that its performance can be influenced by the amount of training, the different configurations and the complexity of the user behavior it is modeling. We suggest that when the situation allows, one should use WorkLog$_\alpha$, and when user behavior is more complex, one should use WorkLog$_\beta$ and tune it to their needs.

6.1 Future Work

We note that a HMM is a generative model. In light of recent developments in machine learning and language models, generative models become more sophisticated and can model more complex data. It will be interesting to see how we can instead leverage state-of-the-art generative models, replacing HMMs. For example, one can use an LSTM to approximate the hidden states of the interleaved HMM. This way, more complex state representations can be learned and potentially achieve better performance.

7 Conclusion

Following the works of [28, 18, 5], we have developed an approach to generate workloads by training an interleaved HMM on small samples of event logs. After an extensive evaluation of two different variants of our approach, WorkLog$_\alpha$ and WorkLog$_\beta$, we have found that both variants can generate realistic logs, and depending on the situation, one can adapt our approach to achieve better performance. We highlight the possibility of extending our approach by leveraging state-of-the-art generative models.

References

[1] Atlassian. [n. d.] Calculating the cost of downtime. en. Retrieved Dec. 11, 2023 from https://www.atlassian.com/incident-management/kpis/cost-of-downtime.

[2] Jeff Barr. 2023. Prime day 2023 powered by AWS - all the numbers. (Aug. 2, 2023). Retrieved Dec. 10, 2023 from https://aws.amazon.com/blogs/aws/prime-day-2023-powered-by-aws-all-the-numbers/.

[3] [SW] James Bradbury et al., jax: Composable transformations of Python+NumPy programs: differentiate, vectorize, JIT to GPU/TPU, and more version 0.3.13, 2018. Github. URL: https://github.com/google/jaxRetrieved Nov. 22, 2023 from.

[4] Axel Busch, Qais Noorshams, Samuel Kounev, Anne Koziolek, Ralf Reussner, and Erich Amrehn. 2015. Automated workload characterization for I/O performance analysis in virtualized environments. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE'15: ACM/SPEC International Conference on Performance Engineering (Austin Texas USA). ACM, New York, NY, USA, (Jan. 31, 2015). DOI: 10.1145/2668930.2688050.

[5] Jinfu Chen, Weiyi Shang, Ahmed E Hassan, Yong Wang, and Jiangbin Lin. 2019. An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (San Diego, CA, USA). IEEE, (Nov. 2019), 669–681. ISBN: 9781728125084. DOI: 10.1109/ase.2019.00068.

[6] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2016. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks. *IEEE Transactions on Software Engineering*, 42, 1148–1161, 12. DOI: 10.1109/TSE.2016.2553039.

[7] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. en. *Psychological bulletin*, 114, (Nov. 1993), 494–509, 3, (Nov. 1993). DOI: 10.1037/0033-2909.114.3.494.

[8] [n. d.] Cloud computing. en. Retrieved Dec. 11, 2023 from https://www.statista.com/study/15293/cloud-computing-statista-dossier.

[9] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. 2005. Capturing, indexing, clustering, and retrieving system history. en. *ACM SIGOPS Operating Systems Review*, 39, (Oct. 20, 2005), 105–118, 5, (Oct. 20, 2005). DOI: 10.1145/1095809.1095821.

[10] Jacob Cohen. 2013. *Statistical Power Analysis for the behavioral sciences*. en. Routledge, London, England, (May 13, 2013). 579 pp. ISBN: 9781134742707. https://books.google.at/books?id=2v9zDAsLvA0C.

[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central. In *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles (Shanghai China). ACM, New York, NY, USA, (Oct. 14, 2017). DOI: 10.1145/3132747.3132772.

[12] Alireza Haghdoost, Weiping He, Jerry Fredin, and David H C Du. 2017. On the accuracy and scalability of intensive {I/O} workload replay. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 315–328. ISBN: 9781931971362. Retrieved Dec. 11, 2023 from https://www.usenix.org/conference/fast17/technical-sessions/presentation/haghdoost.

[13] Ahmed Hassan, Daryl Martin, Parminder Flora, Paul Mansfield, and Dave Dietz. 2008. An Industrial Case Study of Customizing Operational Profiles Using Log Compression. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, 713–723. DOI: 10.1145/1368088.1379445.

[14] [SW] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee, flax: Flax is a neural network library for JAX that is designed for flexibility version 0.7.5, 2023. Github. URL: https://github.com/google/flax Retrieved Nov. 22, 2023 from.

[15] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv [cs.LG]*, (Dec. 22, 2014). Retrieved Nov. 8, 2023 from http://arxiv.org/abs/1412.6980 eprint: 1412.6980 (cs.LG).

[16] Niels Landwehr. 2008. Modeling interleaved hidden processes. In *Proceedings of the 25th international conference on Machine learning - ICML '08*. the 25th international conference (Helsinki, Finland). ACM Press, New York, New York, USA. ISBN: 9781605582054. DOI: 10.1145/1390156.1390222.

[17] Dan Luu. [n. d.] post-mortems: A collection of postmortems. en. Retrieved Dec. 11, 2023 from https://github.com/danluu/post-mortems.

[18] Ariana Minot and Yue M Lu. 2014. Separation of interleaved Markov chains. In *2014 48th Asilomar Conference on Signals, Systems and Computers*. 2014 48th Asilomar Conference on Signals, Systems and Computers (Pacific Grove, CA, USA). IEEE, (Nov. 2014), 1757–1761. ISBN: 9781479982974. DOI: 10.1109/acssc.2014.7094769.

[19] Thomson Reuters. 2022. Taylor Swift fans stormed Ticketmaster. The result was outages, delays and outrage. en. (2022). Retrieved Dec. 10, 2023 from https://www.cbc.ca/news/entertainment/ticketmaster-taylor-swift-tour-1.6653249.

[20] Shlomo S Sawilowsky. 2009. New effect size rules of thumb. *Journal of modern applied statistical methods: JMASM*, 8, (Nov. 1, 2009), 597–599, 2, (Nov. 1, 2009). DOI: 10.22237/jmasm/1257035100.

[21] Bumjoon Seo, Sooyong Kang, Jongmoo Choi, Jaehyuk Cha, Youjip Won, and Sungroh Yoon. 2014. IO Workload Characterization Revisited: A Data-Mining Approach. *IEEE Transactions on Computers*, 63, 3026–3038, 12. DOI: 10.1109/TC.2013.187.

[22] Weiyi Shang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2015. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE'15: ACM/SPEC International Conference on Performance Engineering (Austin Texas USA). ACM, New York, NY, USA, (Jan. 31, 2015). ISBN: 9781450332484. DOI: 10.1145/2668930.2688052.

[23] Jim Summers, Tim Brecht, Derek Eager, and Alex Gutarin. 2016. Characterizing the workload of a netflix streaming video server. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016 IEEE International Symposium on Workload Characterization (IISWC) (Providence, RI, USA). IEEE, (Sept. 2016). DOI: 10.1109/iiswc.2016.7581265.

[24] Mark D Syer, Weiyi Shang, Zhen Ming Jiang, and Ahmed E Hassan. 2017. Continuous validation of performance test workloads. en. *Automated software engineering*, 24, (Mar. 2017), 189–231, 1, (Mar. 2017). DOI: 10.1007/s10515-016-0196-8.

[25] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. 2018. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction-a model-driven approach for session-based application systems. en. *Software & Systems Modeling*, 17, 443–477, 2. DOI: 10.1007/s10270-016-0566-5.

[26] E J Weyuker and F I Vokolos. 2000. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26, 1147–1156, 12. DOI: 10.1109/32.888628.

[27] Huafeng Xi, Jianfeng Zhan, Zhen Jia, Xuehai Hong, Lei Wang, Lixin Zhang, Ninghui Sun, and Gang Lu. 2011. Characterization of real workloads of web search engines. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 2011 IEEE International Symposium on Workload Characterization (IISWC) (Austin, TX, USA). IEEE, (Nov. 2011). DOI: 10.1109/iiswc.2011.6114193.

[28] Neeraja J Yadwadkar, Chiranjib Bhattacharyya, K Gopinath, Thirumale Niranjan, and Sai Susarla. 2010. Discovery of application workloads from network file traces. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*. Retrieved Dec. 11, 2023 from https://www.usenix.org/conference/fast-10/discovery-application-workloads-network-file-traces.

[29] Jianchen Zhao, Yiming Tang, Sneha Sunil, and Weiyi Shang. 2023. Studying and complementing the use of identifiers in logs. en. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (Taipa, Macao). IEEE, (Mar. 1, 2023), 97–107. ISBN: 9781665452786. DOI: 10.1109/saner56733.2023.00019.