Addis Ababa Science and Technology University

Department of Electrical and Computer Engineering

Stochastic Traffic Modeling for Cloud Resource Allocation

Trace-Driven Simulation with Poisson Baselines and an MDP Autoscaler

Course: Stochastic and Random Processes

Instructor: Dr. Sultan

Prepared by:

Kaleab Tadesse
ID: FTP0848/14

January 31, 2026

Abstract

Cloud resource allocation must balance cost and performance under uncertain and bursty workloads. This project models cloud traffic using stochastic methods and optimizes autoscaling decisions using a Markov Decision Process (MDP) formulation. We build a trace-driven workload from the Google Cluster Trace (2011) by extracting task arrivals, CPU/memory requests, and runtimes from raw event logs. We implement an event-driven simulator in Python to evaluate (i) static provisioning, (ii) a queue/backlog threshold autoscaler, and (iii) an MDP-based autoscaler trained on an aggregated backlog model. On the busiest 2-hour trace window (112,434 tasks), the MDP policy achieves markedly lower waiting times and SLA violation rates compared to a fixed-capacity baseline, at a comparable VM-hour cost to heuristic autoscaling. We additionally apply the same trace-driven simulation pipeline to an Alibaba OpenB pod trace (Kaggle), selecting the busiest 24-hour window (681 pods) due to the dataset's lower event density. This provides a second-dataset validation of the cost–performance trade-offs observed on Google.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing platforms provision CPU and memory resources on demand. Workloads are highly dynamic, so providers must choose resource allocations that satisfy performance goals (e.g., low queueing delay) while minimizing operational cost (e.g., VM-hours). This motivates (1) stochastic modeling of uncertain arrivals and demands, and (2) principled control methods such as Markov Decision Processes (MDPs).

1.1 Project goals and scope

This project focuses on:

- building a **trace-driven simulation** using the Google Cluster Trace dataset;
- using **Poisson processes** as a baseline stochastic traffic model;
- formulating and evaluating an **MDP autoscaling** policy;
- comparing policies under a cost–performance trade-off (VM-hours vs delay/SLA).

1.2 Key simplifications (time-constrained project)

To deliver an end-to-end system within a tight project schedule and laptop compute constraints, we apply the following simplifications while preserving the core trade-off:

1. **Homogeneous VM model:** all VMs have normalized capacities $C^{cpu} = 1$, $C^{mem} = 1$.
2. **Requests-based admission:** tasks consume requested CPU/memory rather than actual usage (consistent with many scheduling approaches).
3. **Clean lifecycle filtering:** we keep tasks with a single unambiguous `SUBMIT`→`SCHEDULE`→`FINISH` lifecycle and ignore evictions/retries/preemptions.
4. **Simple placement:** FIFO queue with a first-fit placement across VMs.
5. **Control discretization:** autoscaler decisions occur every $\Delta = 60$ seconds.

Chapter 2

Background

2.1 Stochastic workload modeling and Poisson processes

A common baseline for random arrivals is the Poisson process with rate λ, implying exponentially distributed inter-arrival times and independent increments. Poisson models are mathematically tractable and often used as a first-order approximation in queueing analysis. However, production traces frequently exhibit burstiness and correlations, motivating trace-driven simulation and/or non-homogeneous extensions.

2.2 Markov Decision Processes (MDP)

An MDP is defined by $\langle S, A, P, R, \gamma \rangle$, where S is the set of states, A the set of actions, P transition probabilities, R rewards, and $\gamma \in (0, 1)$ a discount factor. In autoscaling, the state summarizes current load and capacity; actions add/remove VMs; and the reward encodes the cost–performance objective.

Chapter 3

Dataset: Google Cluster Trace (2011)

3.1 Source

We use the Google Cluster Trace (2011, version 2), which records task lifecycle events and resource requests across a large production cluster. The dataset is provided as sharded `task_events` CSV files in Google Cloud Storage [goo].

3.2 Raw schema used

From `task_events`, we use:

- time (microseconds since trace start),
- job ID, task index,
- event type (submit/schedule/finish),
- CPU request and memory request (normalized fractions).

3.3 Preprocessing pipeline

We convert raw event logs into a compact task table:

$$(\text{arrival\_time},\ \text{runtime},\ cpu\_req,\ mem\_req).$$

Definitions:

- arrival time t_a: time of `SUBMIT` event,
- start time t_s: time of first `SCHEDULE` after submit,
- finish time t_f: time of first `FINISH` after start,
- runtime $d = t_f - t_s$.

We filter tasks to those with $d > 0$, $d \leq 86400$ seconds, and $0 < cpu\_req, mem\_req \leq 1$.

3.3.1 Trace subset and window selection

To keep processing manageable, we download a subset of consecutive `task_events` shards, extract clean tasks, and select the busiest 2-hour window by maximizing arrivals in a sliding 2-hour window (1-minute bins). The selected busiest window contains 112,434 tasks.

3.3.2 Summary statistics of the selected window

For the busiest 2-hour window:

- number of tasks: 112,434,
- mean runtime: ≈ 734.6 seconds,
- mean CPU request: ≈ 0.0477,
- mean memory request: ≈ 0.0329.

Chapter 4

System and Simulation Model

4.1 Resource model

We simulate a pool of identical VMs, each with normalized capacity:

$$C^{cpu} = 1, \quad C^{mem} = 1.$$

A task i requests (c_i, m_i) and occupies those resources for its runtime d_i.

4.2 Scheduling model

Tasks arrive at t_a and join a FIFO queue if they cannot be placed immediately. A task is placed on the first VM that can accommodate its CPU and memory requests (first-fit). When a task finishes, it releases resources.

4.3 Simulation approach

We implement a discrete-event simulation:

- arrival events add tasks to the queue,
- completion events release resources and trigger additional placements,
- autoscaling decisions occur every $\Delta = 60$ seconds.

4.4 Cost and performance metrics

4.4.1 Cost

We measure cost as the time integral of active VM count:

$$\text{VM-seconds} = \int_0^T k(t)\, dt, \qquad \text{VM-hours} = \frac{\text{VM-seconds}}{3600}.$$

Note: the simulation runs past the 2-hour arrival window until all tasks complete, so VM-hours can exceed $2 \times k$.

4.4.2 Performance and SLA metrics

We report:

- mean waiting time,
- p95 and p99 waiting time,
- SLA60 violation rate: $P(W > 60s)$,
- SLA120 violation rate: $P(W > 120s)$.

Chapter 5

Autoscaling Policies

5.1 Static provisioning

A fixed number of VMs k remain active for the entire simulation.

5.2 Threshold autoscaling

Every $\Delta = 60$ seconds we compute an aggregated queue backlog signal:

$$Q(t) = \sum_{i \in \text{queue}} \max(c_i, m_i) \cdot d_i,$$

interpretable as queued *resource-seconds* of dominant demand.

If $Q(t)$ exceeds an upper threshold, we scale up by a step size; if it falls below a lower threshold, we scale down by a step size. Step sizes are used to ensure responsiveness under bursty demand.

5.3 MDP-based autoscaling (aggregated model)

A full MDP state including per-VM allocations and remaining runtimes is too large for a short project. Instead, we use an aggregated MDP approximation:

- State: $s = (k, \text{bin}(Q))$, where k is active VMs and $\text{bin}(Q)$ discretizes queued work.
- Actions: $a \in \{-100, -50, 0, +50, +100\}$ VMs, clipped to $[k_{\min}, k_{\max}]$.
- Dynamics: $Q_{t+1} = \max(0, Q_t + W_t^{in} - k_t \Delta)$, where W_t^{in} is sampled from trace-derived arrivals aggregated per control interval.
- Reward: negative weighted sum of VM cost, backlog, and action magnitude.

We learn the policy using tabular Q-learning on the aggregated MDP, then evaluate it in the detailed event-driven simulator.

Chapter 6

Experiments

6.1 Experimental setup

We evaluate on the busiest 2-hour window extracted from the Google trace. Control interval is $\Delta = 60$ seconds. The MDP training estimates capacity bounds $k_{\min} = 186$ and $k_{\max} = 1200$ from trace-derived arrival work statistics.

6.2 Policies evaluated

- Static provisioning (baseline),
- Threshold autoscaling (heuristic),
- MDP autoscaling (learned policy).

Chapter 7

Results and Discussion

7.1 Experimental context and metrics

We evaluate three allocation strategies on the busiest 2-hour window extracted from the Google Cluster Trace (112,434 tasks). The arrival window is 2 hours, but the simulation continues until all tasks complete; therefore, cost is measured as VM-hours over the full completion horizon. Performance is evaluated via waiting-time percentiles (p95/p99) and SLA violation rates, where SLA60 denotes $P(W > 60\text{s})$ and SLA120 denotes $P(W > 120\text{s})$.

7.2 Overall cost–SLA trade-off

Figure 7.1 summarizes the cost–SLA60 trade-off. The static provisioning sweep forms a cost–SLA curve: increasing capacity (higher VM-hours) reduces the probability that tasks wait longer than 60 seconds. Autoscaling policies shift the operating point by adapting capacity in response to bursts.

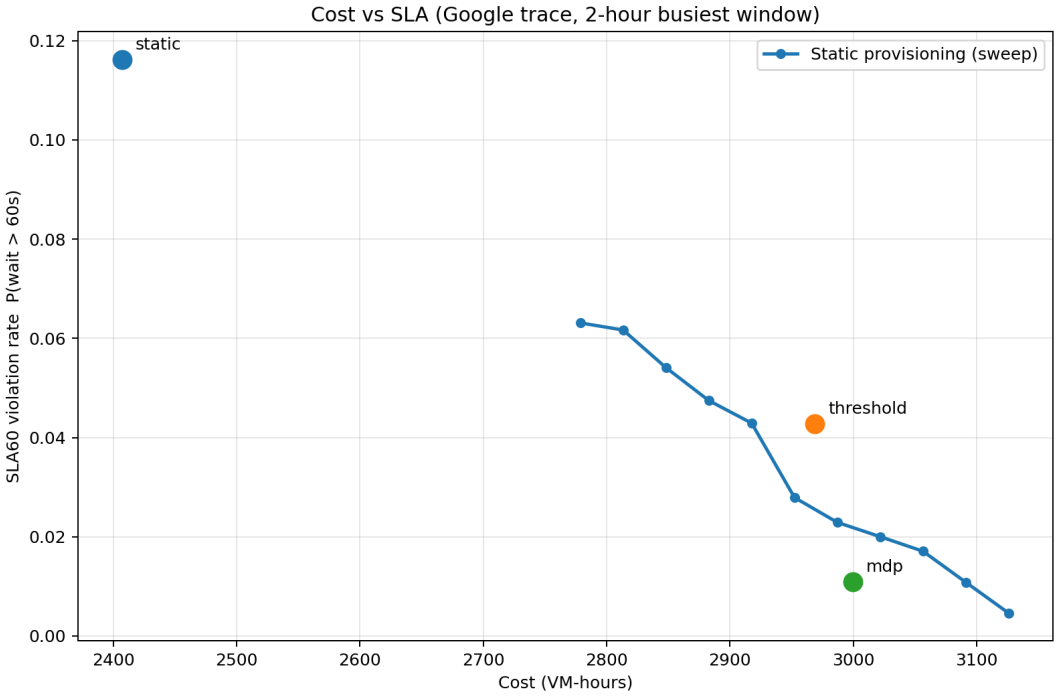

Figure 7.1: Cost vs SLA60 on the busiest 2-hour Google trace window. The static sweep forms a cost–SLA curve; threshold and MDP appear as operating points.

7.3 Main quantitative results

Table 7.1 reports cost and delay metrics for the three primary policies.

Table 7.1: Trace-driven simulation results (Google trace, busiest 2-hour window; 112,434 tasks).

Policy	VM-hours	Mean wait (s)	p95 (s)	p99 (s)	SLA60	SLA120
Static (k=693)	2407.0	47.23	442.50	685.41	0.1162	0.1146
Threshold	2968.7	7.81	51.86	105.39	0.0428	0.0000
MDP	2999.7	2.33	10.95	82.04	0.0109	0.0042

Overall, the fixed-capacity baseline (k=693) is cheapest among these three configurations but suffers substantially worse tail delay and SLA violations during burst periods. Threshold autoscaling reduces both p95/p99 delays and SLA60 compared to static, at a higher VM-hour cost. The MDP policy achieves the lowest waiting times and lowest SLA60 violation rate, at a cost comparable to the threshold heuristic in this configuration.

7.4 Matched-SLA comparisons (fair static baselines)

A single static baseline can be misleading because static provisioning represents a family of policies parameterized by k. To ensure fair comparisons, we perform matched-SLA baselining: for each autoscaling policy we select a static capacity that achieves approximately the same SLA60 violation rate, then compare cost and tail latency.

Table 7.2: Matched-SLA comparisons against static provisioning (Google trace, busiest 2-hour window).

Policy / Baseline	VM-hours	SLA60	p99 wait (s)	Notes
Threshold autoscaling	2968.7	0.0428	105.39	Heuristic control point
Static (k=840)	2917.6	0.0429	180.55	Matches threshold SLA60
MDP autoscaling	2999.7	0.0109	82.04	Learned control point
Static (k=890)	3091.3	0.0108	66.02	Matches MDP SLA60

Note: For high static capacities, p95 waiting time can become 0 because at least 95% of tasks start immediately. Therefore, SLA violation rates and p99 waiting time provide a more informative fairness comparison.

At approximately the same SLA60 as the threshold autoscaler, Static(k=840) is slightly cheaper in VM-hours but exhibits substantially worse tail latency (p99). At approximately the same SLA60 as the MDP policy, Static(k=890) requires more VM-hours, while offering slightly lower p99 in this configuration. This indicates that adaptive policies can reduce cost at a given SLA target, and that tail latency depends on how strongly the policy objective penalizes extreme backlog.

7.5 Policy behavior over time

Figures 7.2–7.4 visualize the mechanism behind the metrics. Static provisioning keeps capacity constant, so bursts create larger backlog that increases tail delays. Threshold and MDP autoscaling increase capacity during burst periods, reducing queueing and improving SLA metrics.

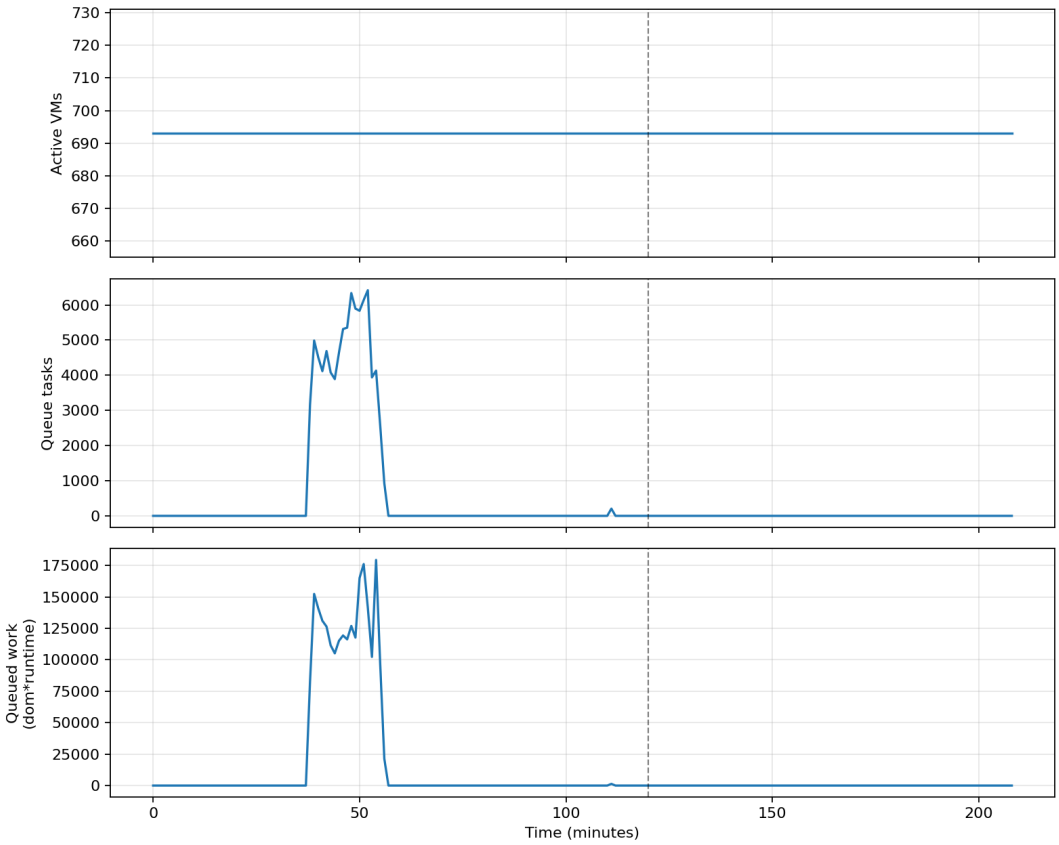

Figure 7.2: Static provisioning: constant VM capacity. Bursty arrivals create larger queued backlog and higher tail delays.

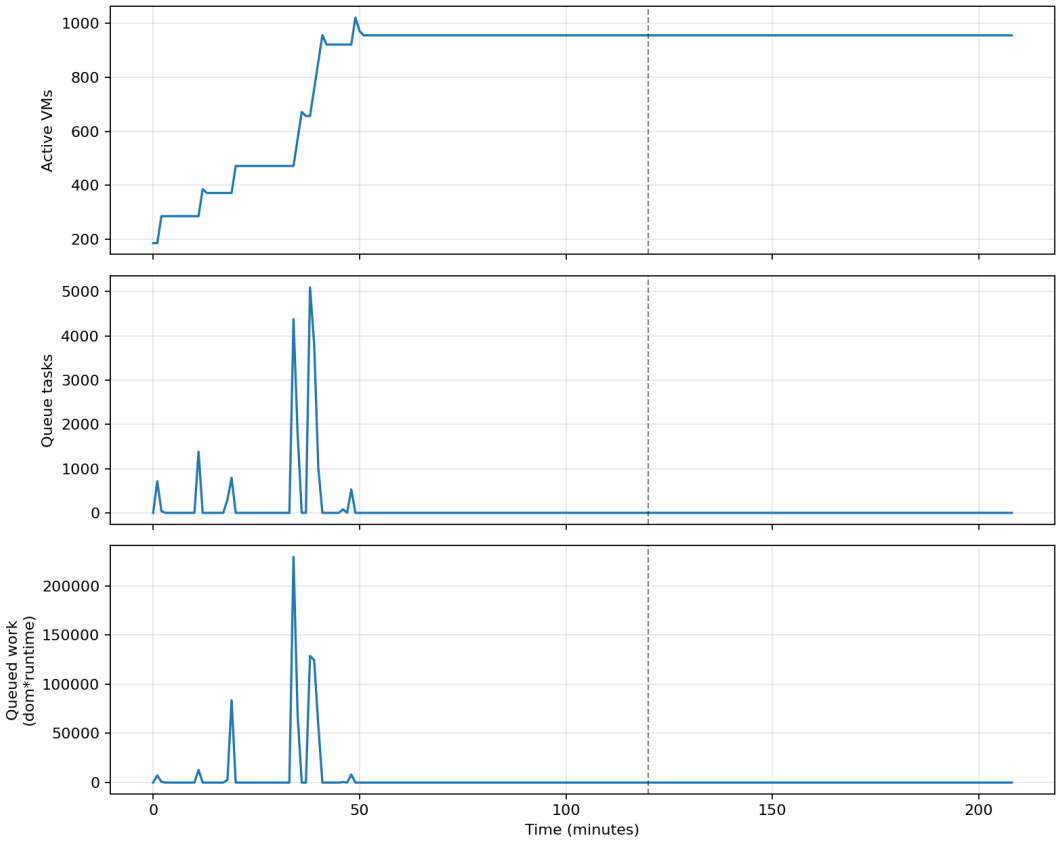

Figure 7.3: Threshold autoscaling: VMs increase in response to queued work and reduce backlog during bursts.

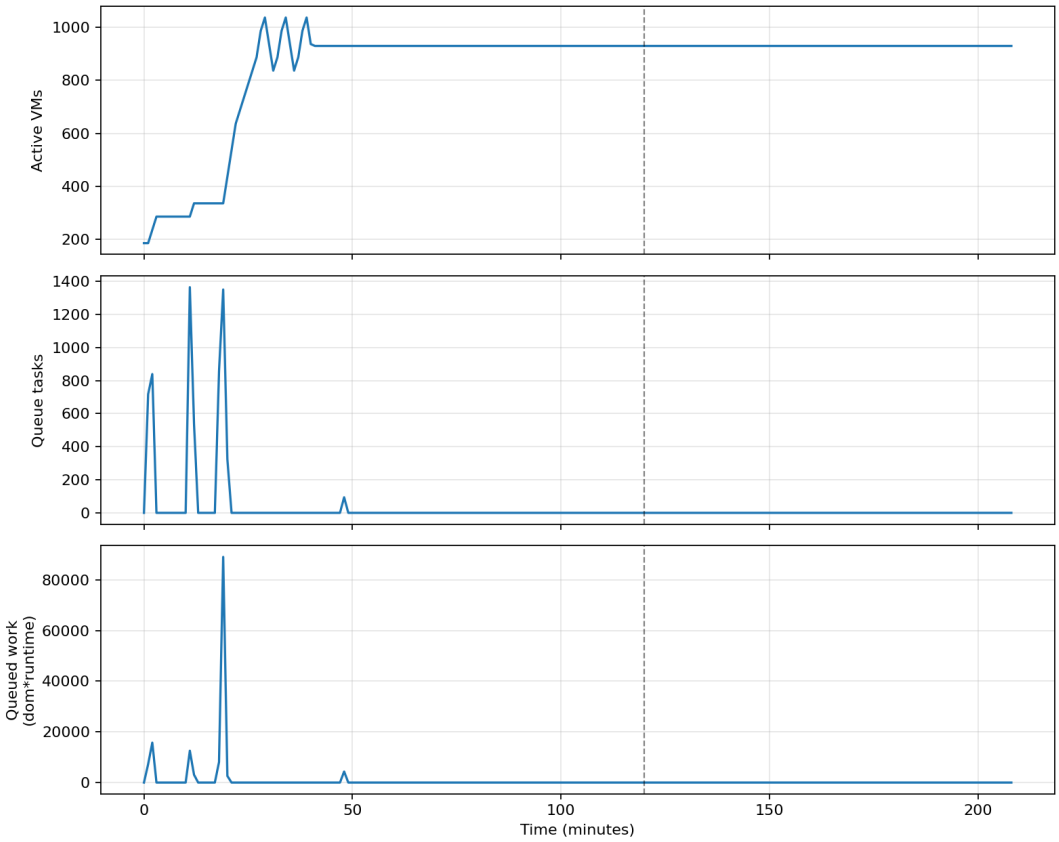

Figure 7.4: MDP autoscaling: learned actions respond more aggressively to backlog, reducing queueing and SLA violations.

Chapter 8

Limitations

- **Homogeneous capacity:** real clusters have heterogeneous machines and VM types.
- **Requests vs usage:** we use requested CPU/memory, not actual utilization.
- **Ignored failure modes:** retries, evictions, and preemptions are filtered out.
- **Simplified scheduler:** FIFO + first-fit does not reproduce production scheduling constraints.
- **Aggregated MDP:** the MDP state compresses dynamics into a single queued-work signal.

Chapter 9

Dataset: Alibaba OpenB Pod Trace (Kaggle)

9.1 Source

To satisfy the assignment requirement of using Alibaba traces, we use the *Alibaba OpenB pod trace* available via Kaggle [ali]. The dataset contains pod-level resource requests and lifecycle timestamps, enabling conversion to the same task format used in our simulator.

9.2 Raw schema used

We use the following fields from the pod list tables (e.g., `openb_pod_list_default.csv`):

- `cpu_milli` (CPU request in millicores),
- `memory_mib` (memory request in MiB),
- `creation_time`, `scheduled_time`, `deletion_time` (timestamps),

and we use node capacity fields from `openb_node_list_all_node.csv`: `cpu_milli` and `memory_mib`.

9.3 Preprocessing and normalization

We convert each pod record into:

$$(\text{arrival\_time},\ \text{runtime},\ \textit{cpu\_req},\ \textit{mem\_req}).$$

We set:

- arrival time $t_a =$ `creation_time`,
- start time $t_s =$ `scheduled_time` (fallback to `creation_time` if missing/0),
- finish time $t_f =$ `deletion_time`,
- runtime $d = t_f - t_s$.

To match our homogeneous VM simulator, we normalize CPU and memory requests by node capacities observed in the trace: $C^{cpu} = 32000$ millicores and $C^{mem} = 262144$ MiB (from `openb_node_list_all_node.csv`), so that $cpu\_req, mem\_req \in (0, 1]$. We filter to pods with positive runtime and requests that fit within a single normalized VM.

9.4 Window selection

Unlike the Google trace, this Alibaba OpenB dataset is relatively sparse over time. A 2-hour window contained too few events for stable autoscaling evaluation, so we select the busiest 24-hour window using a sliding-window arrival count. The selected busiest 24-hour window contains 681 pods.

Chapter 10

Results on Alibaba Trace (OpenB Pods)

10.1 Main quantitative results

Table 10.1 summarizes cost and performance on the busiest 24-hour Alibaba window (681 pods). Because many pods start immediately when sufficient capacity is provisioned, p95 can become 0; therefore we also report SLA violation rates and p99.

Table 10.1: Trace-driven simulation results (Alibaba OpenB pods, busiest 24-hour window; 681 pods).

Policy	VM-hours	Mean wait (s)	p95 (s)	p99 (s)	SLA60	SLA120
Static (k=83)	2596.7	0.00	0.00	0.00	0.0000	0.0000
Threshold	144.6	35.57	198.00	889.80	0.1175	0.0808
MDP	202.4	1.46	0.00	80.40	0.0147	0.0000

The threshold autoscaler achieves a large cost reduction compared to static provisioning, but incurs higher tail latency and SLA violations. The MDP policy reduces SLA60 substantially (from 11.75% to 1.47%) and lowers p99 waiting time, at a moderate increase in VM-hours relative to the threshold baseline.

10.2 Cost–SLA trade-off

Figure 10.1 plots VM-hours versus SLA60 for a static capacity sweep and overlays the threshold and MDP operating points.

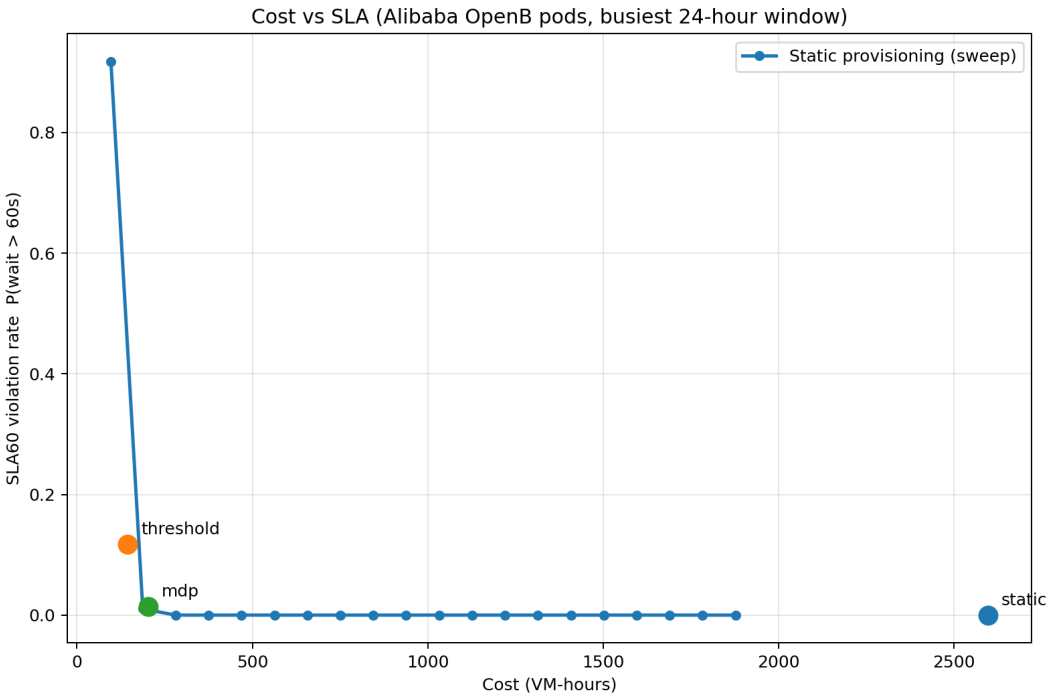

Figure 10.1: Cost vs SLA60 on the Alibaba OpenB pod trace (busiest 24-hour window).

10.3 Temporal behavior

Figures 10.2–10.4 illustrate how the policies respond over time.

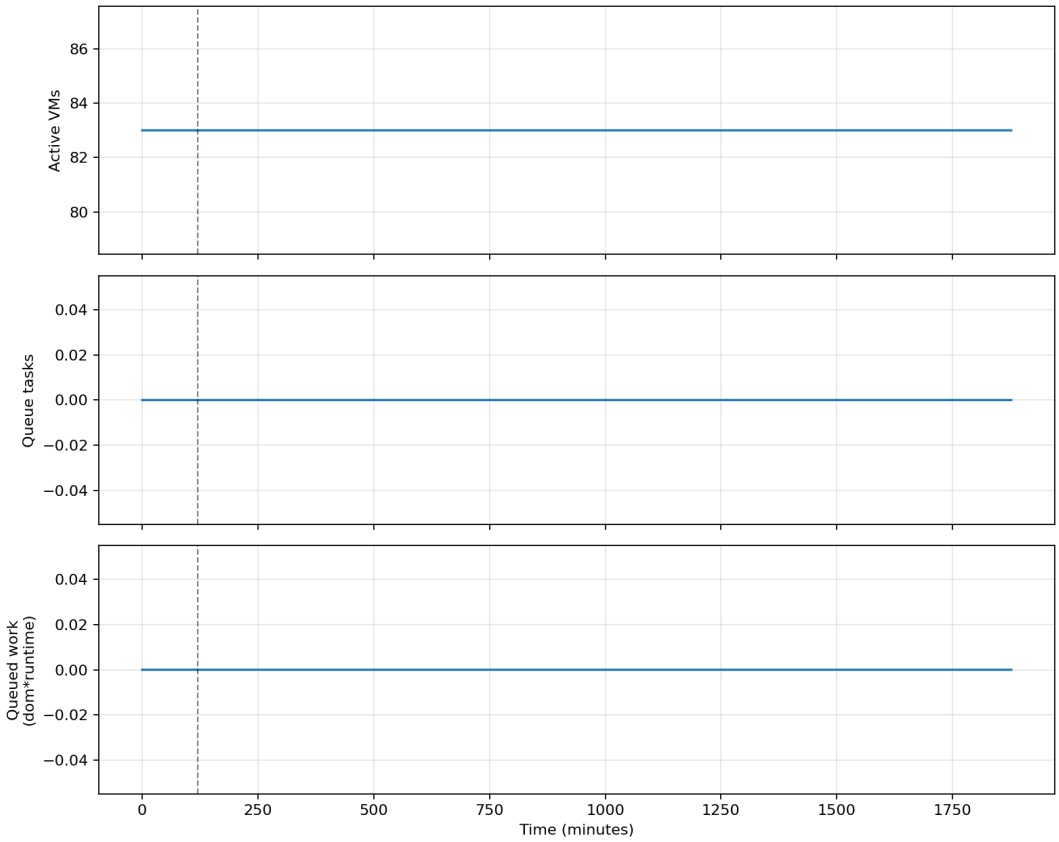

Figure 10.2: Alibaba: Static provisioning (fixed VM capacity).

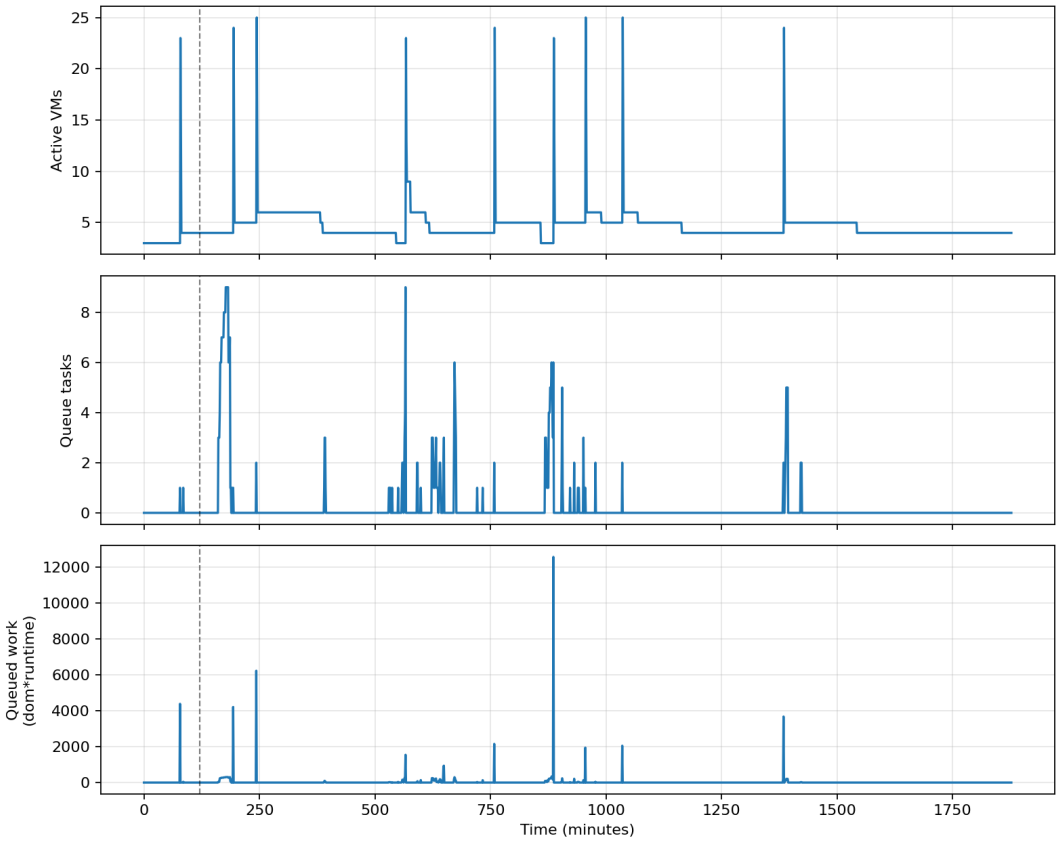

Figure 10.3: Alibaba: Threshold autoscaling response over time.

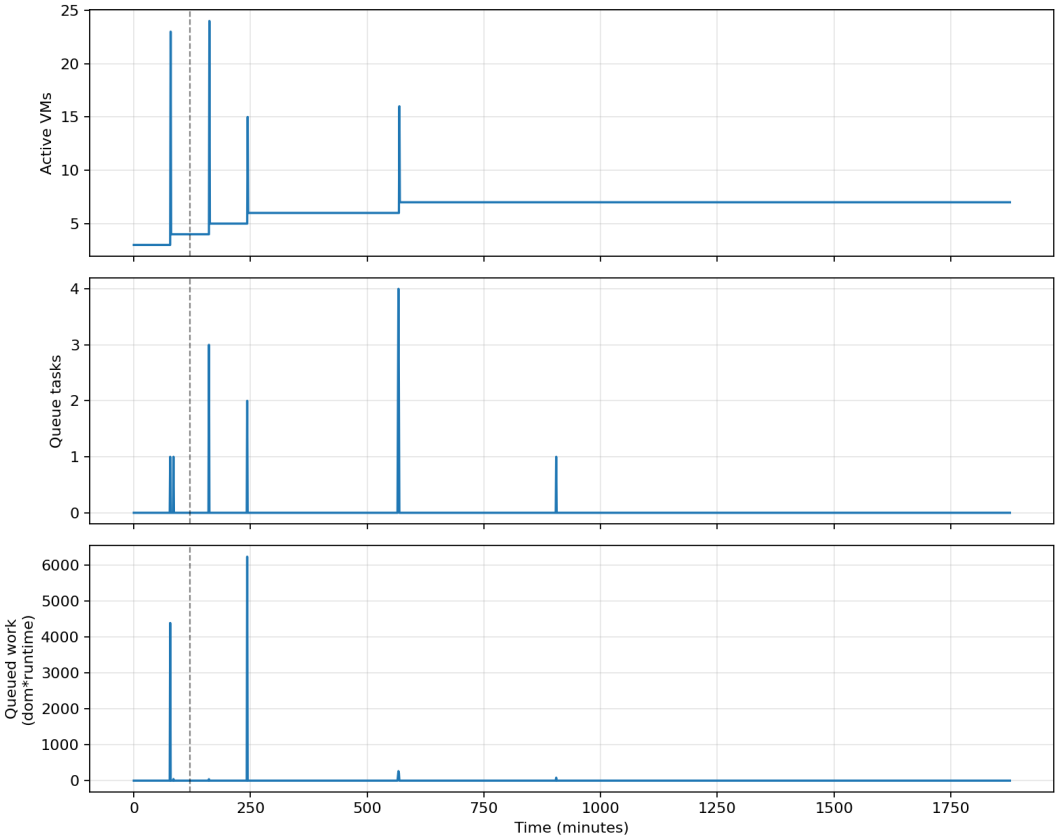

Figure 10.4: Alibaba: MDP autoscaling response over time.

Chapter 11

Conclusion

This project demonstrates how stochastic modeling and MDP-based decision-making can guide cloud resource allocation under uncertainty. Using a trace-driven workload from the Google Cluster Trace and an event-driven simulator, we quantify the cost–performance trade-offs between static provisioning, heuristic threshold scaling, and an MDP autoscaler. Results show that autoscaling policies can dramatically reduce waiting times and SLA violations during bursts, and an MDP policy can further reduce SLA60 at comparable cost by adapting capacity to the workload over time.

Bibliography

alibaba_full (openb pod trace) – kaggle dataset. https://www.kaggle.com/datasets/nimishgsk17/alibaba-full. License: unknown. Accessed 2026-01-31.

Google cluster data (2011): Clusterdata2011_2. https://github.com/google/cluster-data. Accessed 2026-01-28.

Appendix A

Reproducibility Notes

- Data extraction: DuckDB SQL to build a clean task table from raw `task_events`.
- Simulator: Python discrete-event engine with CPU+memory constraints.
- Figures: generated by scripts in `src/` and saved to `figures/`.