INTERACTIVE CONTROL ENGINEERING
Principles and Applications

Volume 8 Batch, Run-to-Run, and Hybrid Control

CHAPTER 14

Integer Servers, Boot Delays and Retry Storms in a Cloud Autoscaling Simulator

Pages 330–353

Edited by John D. Hedengren

How to cite this chapter. "Integer Servers, Boot Delays and Retry Storms in a Cloud Autoscaling Simulator," in *Batch, Run-to-Run, and Hybrid Control*, John D. Hedengren, Ed., vol. 8 of *Interactive Control Engineering: Principles and Applications*. Advanced Process Solutions, LLC, 2026, ch. 14, pp. 330–353.

The page numbers in this file are those of the printed volume.

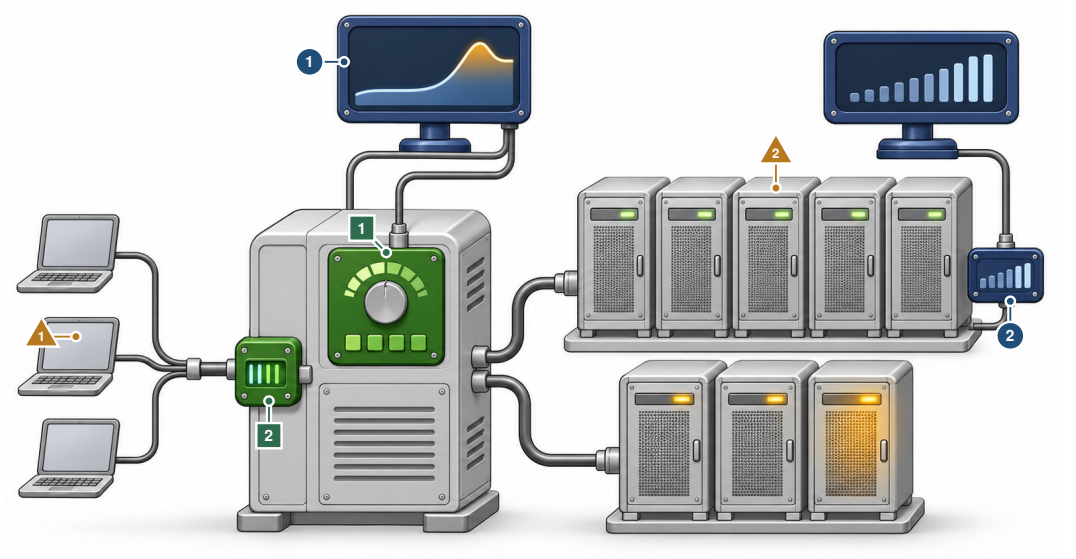
The app for this chapter
https://apmonitor.com/apps/cloud-autoscaling/

The volume online: contents, other chapters and citation records
https://apmonitor.com/apps/books/v08-batch-hybrid/#cloud-autoscaling

Copyright © 2026 Advanced Process Solutions, LLC. All rights reserved. Produced by Advanced Process Solutions, LLC.

CHAPTER 14

Integer Servers, Boot Delays and Retry Storms in a Cloud Autoscaling Simulator



The controller moves 1 the autoscaler's order of whole servers, which boot before they serve
2 admission control at the load balancer

The controller reads 1 the minute's 99th-percentile latency against its objective
2 the busy fraction of the ready servers, the metric the autoscaler reads

What disturbs it 1 a flash crowd of users 2 a preemption that removes the spot servers

Requests flow from the users through the balancer to the ready servers, so notice that the booting servers on the lower shelf serve nothing until their boot ends, which is why servers ordered in a crowd arrive minutes after it began. Illustration, not to scale.

IN PLAIN TERMS Ordering Servers That Arrive Late

The main idea. A web service runs on a group of computers. A program watches how busy they are and adds or removes whole computers as the traffic rises and falls.

The challenge. A new computer needs about three minutes to start before it can answer anyone. A sudden crowd can come and go before that help arrives. When answers are slow, users press again, and their repeats can keep the service jammed after the crowd has left.

Why it matters. Shops, banks and news sites need a service that stays quick at the busiest moment, which is when extra computers are slowest to help.

THREE WAYS TO READ THIS CHAPTER

Explore	12 min	the first page, the history in Section 14.2, the fact in Section 14.4, the two try-its and the fact in Section 14.5, and the key points
Study	50 min	the whole chapter with its three pause points and the twelve exercises at its end
Practice	20 min	the first page, the results (Section 14.5), Technology in Practice (Section 14.7), Limitations (Section 14.8) and the two open challenges

The control question An autoscaler orders whole servers that serve only after a boot, on a metric that lags and cannot read above full, while the objective is a latency percentile that stays flat until the servers are nearly full. When does capacity ordered by feedback arrive too late, and what keeps the objective when it does?

Learning objectives

- Compute the 99th-percentile latency from the declared queueing term, and locate the knee and the objective on it.
- Evaluate the platform's desired-replica rule on given readings, and explain why a saturated CPU reading limits each decision.
- Compare the rule, a PI loop and a forecast floor on a ramp by breach minutes and bill.
- Explain how client retries hold a service in overload after its cause has gone, and how a retry budget and admission control break the loop.
- Judge when no autoscaler can meet an objective without shedding.

Prerequisites Sections 1.4.7 (hysteresis and dwell), 1.4.8 (integer actuators), 1.6.2 (the PI law as implemented) and 1.6.4 (scenarios and data). Section 1.6.3 is needed only for the predictive controller.

The app https://apmonitor.com/apps/cloud-autoscaling/

See also Volume 5, Chapter 2 for a disturbance shorter than the dead time of the loop that answers it, Volume 4, Chapter 2 for a protective override, and Volume 2, Chapter 8 for metering a queue short of its breakdown.

14.1 Introduction

At 20:00 in the fourth case of the simulator of this chapter, the traffic to a web service quadruples in two minutes. The autoscaler sees it in its next published metric and orders servers at once, and the first new server begins to serve at 20:03:45, because every server needs 180 s to boot. The crowd holds for six minutes and is gone by 20:12. The fleet keeps growing after that, to the 200-server quota at 20:16, while only 17.2 % of the users' requests between 20:00 and 20:15 were answered within the clients' 2 s timeout. The service spends 14 minutes above its 300 ms latency objective, and 23.7 % of the users' requests from 19:45 to 21:45 are finally dropped. Nothing in the rule was wrong. The capacity it ordered arrived after the need for it had ended. A pellet grill whose lid is opened for three minutes meets the same difficulty, with fuel that burns after the lid has shut (Volume 5, Chapter 2).

Cloud computing made capacity a quantity that software can order by the second [1], and elastic services commonly carry an autoscaler that measures how busy its servers are and adds or removes them [2, 3]. These loops are feedback controllers in the full sense, with a sensor, a setpoint, an actuator and a disturbance [4], and they inherit the classic difficulties of process control in an unfamiliar vocabulary. The actuator is an integer. It arrives after a dead time set by the boot, which motivated capacity managers that keep servers on rather than pay that delay at the worst moment [5]. The measurement is an average that lags, and a busy fraction cannot read above 100 %. The quantity users care about is a tail percentile of latency [6], and queueing theory says that it stays nearly flat until the servers are close to full and then grows as $1/(1-\rho)$ [7, 8]. A request that fails is often sent again, which turns an overload into extra load [9, 10].

The word elastic encourages a belief that this simulator sets out to correct, namely that capacity in the cloud follows the demand. In this model a rule that keeps the average utilization at 70 % follows the morning ramp of case 2 without a single breach minute, yet the same rule leaves 14 breach minutes in a twelve-minute crowd, and with the fleet capped at 60 servers in case 7 the clients' retries keep the service in overload until 22:10, two hours after the crowd ended. The multi-server queue behind that curve is the standard model of a server farm [8], and the same pairing of a discrete, slow capacity with a queue whose wait explodes near saturation appears in call centers staffed by shifts and in hospital wards that open beds. A block diagram of a proportional loop with a delay shows the lag but none of the knee, the integer rounding, the saturated sensor or the retries.

The literature a control student would read treats these pieces in separate places. Surveys classify autoscaling policies by technique [2, 3], control-theoretic work analyzes one loop at a time [4, 11, 12], and the retry storm appears in operations practice and in the study of metastable failures [9, 10, 13]. This chapter takes one browser simulator, in which a declared queueing model, an integer fleet with a boot delay, the platform's metric pipeline, the published scaling rule and the clients' retries run together, and measures on it how each of the standard remedies (a lower target, a cooldown, a forecast, admission control and a retry budget) changes the breach minutes, the dropped requests and the bill. None of the pieces is new. The rule is the documented algorithm of a public open-source platform [14], the latency term is the Kingman and Sakasegawa approximation [7, 15], and the retry storm and the retry budget are described in the sources cited above. The simulator joins them in a day-long fluid model whose request ledgers close to round-off, and the measurements on it include a paired-seed comparison in which the retry budget removes every storm minute on ten of ten seeds. The app scores the nine cases and the graded day by completion rules (Section 1.6.4).

Section 14.2 follows the domain from admission control to the desired-replica rule. Section 14.3 gives the model of the service: traffic, the request fluid, the latency knee, retries, the integer fleet and the metric pipeline. Section 14.4 gives the autoscalers: the platform's rule, a PI loop, the forecast floor, the predictive controller and the two overrides. Section 14.5 measures the cases and the day, Section 14.6 sorts the evidence, Section 14.7 describes the real metrics pipeline, machine launches and controllers, Section 14.8 states the limitations, and Section 14.9 draws the conclusions. The Exercises at the end of the chapter run the cases.

14.2 From Admission Control to the Desired-Replica Rule

Web servers met feedback control before cloud computing existed. In 2002 classical feedback control was applied to a web server end system for overload protection and per-class response-time and throughput guarantees, with a model of the server and named equivalents of its sensors and actuators [11]. Two years later a book collected the modeling, transfer functions, proportional and PID control and state-space design of computing systems in one text [4]. Capacity in that era was bought in advance, so the loops managed admission and resource shares inside a fixed machine rather than the number of machines.

Elastic computing changed the actuator. A view of cloud computing from Berkeley, published in 2010, named the illusion of capacity on demand and pay for use on a short-term basis among the properties that made the cloud new [1]. Control researchers saw that the new actuator was coarse. A study of automated control in clouds treated the virtual machine count as a discrete actuator and proposed proportional thresholding, a target range for an integral controller that depends on the size of the fleet [12]. Forecasting entered next, with a model-predictive algorithm that forecast the workload and allocated resources ahead of it [16], and a capacity manager for multi-tier data centers kept just enough spare servers on to absorb bursts in the request rate [5]. By 2014 a review could classify the techniques into threshold rules, reinforcement learning, queueing theory, control theory and time-series analysis [2], and a 2018 survey built a taxonomy of web-application autoscalers from the challenges that remained open [3].

> ⧖ HISTORY **2011: Elasticity Gets a Federal Definition**
>
> In September 2011 the National Institute of Standards and Technology published a definition of cloud computing built from five essential characteristics, three service models and four deployment models [17]. One characteristic is rapid elasticity. Capabilities are provisioned and released, in some cases automatically, to scale outward and inward with demand, and to the consumer they often appear to be unlimited and available in any quantity at any time. Another is measured service, a metering capability through which the use of resources is monitored, controlled and reported, typically on a pay-per-use basis [17]. Elasticity is the actuator of this chapter's loop and metering is its sensor. The 200-server quota and the 180 s boot of Table 14.2 are what the appearance of unlimited capacity leaves out.

Table 14.1 Variables by Role, With the Values at the Screenshot Instant of Case 4 (20:06)

Symbol	Role	20:06	Unit
p_{99}	CV, minute 99th percentile, objective 300 ms	91.56	s
u	measured, busy fraction or requests per server	1.00	—
u_{SP}	SP of the autoscaler	0.70	—
n_{rep}	MV, replicas, active plus booting	75	servers
n_{act}, n_{boot}	states, active and booting servers	52, 23	servers
λ_{user}	DV, users' first attempts (truth)	9614	req/s
λ_{retry}	state flow, retries arriving (truth)	18 715	req/s
λ_{off}	measured, offered to the balancer	28 329	req/s
—	refused at the queue limit	23 129	req/s
q	state, backlog (truth)	156 000	requests
f_{out}	output, share timing out (truth)	0.904	—
—	answered in time (truth)	498	req/s
—	bill rate	15.00	$/h
—	DVs, crowd, preemption, slow dependency, deploy, forecast miss	—	events

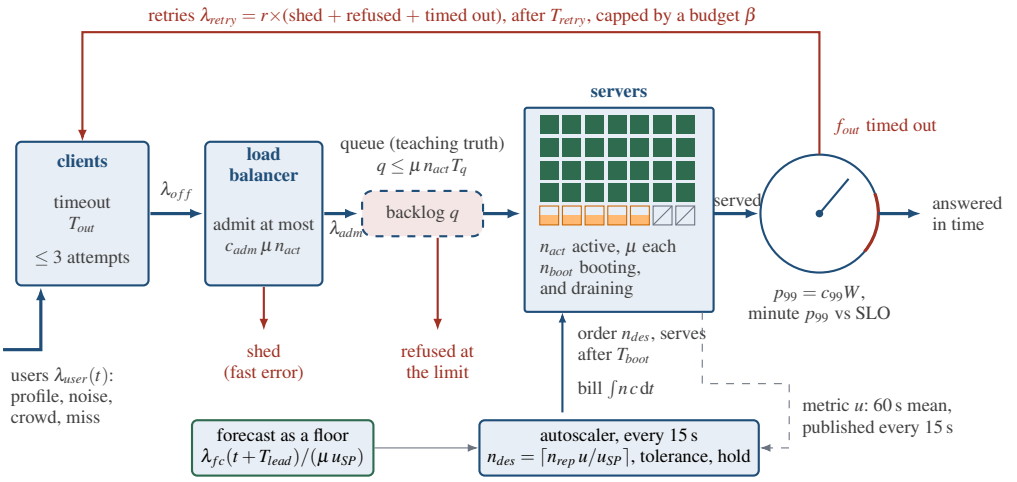
The rule that became the default was the simplest. Container orchestration grew from a decade of internal cluster managers into an open-source platform [18]. The horizontal pod autoscaler of that platform, Kubernetes, multiplies the current replica count by the ratio of the measured metric to its target and rounds up, skips any change within a 10 % tolerance, runs every 15 s, counts pods that are not yet ready as idle when it scales up, holds a scale-down to the highest recommendation of a 300 s stabilization window, and limits a scale-up to doubling or four pods per period [14]. In control terms it is a proportional controller on the load with a deadband, a hold and an integer output. This simulator writes that algorithm out from its public documentation.

What the loop protects changed as well. Large services learned that a tail percentile, not the mean, decides what users experience, because one slow component among many delays a whole request [6], and site-reliability practice turned the percentile into a service-level objective with an error budget, together with rules for handling overload such as refusing work early, capping the attempts per request at three and giving each client a retry budget of 10 % of its requests [9]. The overload that retries sustain after its trigger has gone was later named a metastable failure [10], and recent work analyzes how such a congestive collapse is triggered and why it persists after the trigger is removed [13]. The queueing law under all of it is older. Kingman's 1961 heavy-traffic result gives the mean wait of a single-server queue as $\rho/(1-\rho)$ times the variability and the service time [7], and Sakasegawa's 1977 formula extends the form to several servers [15].

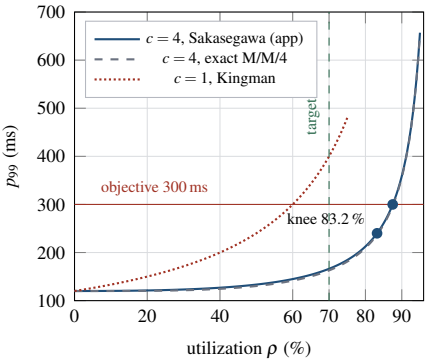
14.3 The Service as Modeled

The simulated system is one stateless web service, drawn left to right as the application's Service view draws it, in Figure 14.1(a). Users send requests at $\lambda_{user}(t)$. Clients wait at most T_{out} for an answer and send a failed request again after T_{retry}, up to three attempts. A load balancer admits at most a share c_{adm} of the active capacity when admission control is on, refuses the rest at once, and holds what it admits in a queue limited to T_q seconds of the fleet's capacity. Servers are integers that pass from ordered to booting to active to draining. The latency of the admitted flow is reported as a per-minute 99th percentile against the objective, and an autoscaler reads the platform's metrics every 15 s and orders servers. The plant integrates in seconds with a step of $\Delta t = 1$ s, and a run starts at the case's clock time.

The quantities that cross the boundary of the loop are classified in Table 14.1. The controlled quantity is the minute's p_{99}, held under an objective as a constraint, while the autoscaler's own setpoint is a utilization target u_{SP} on the metric it reads. The manipulated variable is the integer replica count n_{rep}.



(a)



(b)

Figure 14.1 The autoscaled web service as modeled, and its latency knee. (a) Users at λ_{user} and retries at λ_{retry} arrive as the offered flow λ_{off}. The balancer admits λ_{adm}, sheds the excess over $c_{adm}\mu n_{act}$ when admission control is on, and refuses what would wait beyond $\mu n_{act} T_q$. Active servers (solid squares) serve at μ each, booting servers (half-filled squares) serve after T_{boot}, and drained servers are outlined and crossed. Failures return as retries, and the autoscaler orders n_{des} from the published metric u, with the forecast as a floor. The backlog q is hidden from every controller. (b) p_{99} against utilization from the application's own functions, for four workers per server with the Sakasegawa term the application uses (solid) and with the exact M/M/4 wait (dashed), and for one worker (dotted). The knee at 83.2 % is where queueing doubles the unloaded 120 ms, and the 300 ms objective falls at 87.5 %.

14.3.1 Traffic

The users' traffic is a declared diurnal profile with colored noise, a forecast-miss level and a flash crowd, as shown in Equation (14.1).

$$\begin{aligned}\lambda_{user}(t) = \lambda_{peak}\, D(t_c)\,\big(1+\sigma\,\xi(t)\big) \\ \times L(t)\,\big(1+(X-1)\,F(t-t_X)\big)\end{aligned} \tag{14.1}$$

where t_c is the clock time, D is a periodic Catmull-Rom spline through hourly knots from 0.17 of the peak at 04:00 to 1.0 at 21:00 (1920 and 2400 req/s at 15:00 and 21:00 for $\lambda_{peak} = 2400\,\text{req/s}$, a daily mean of 1574 req/s), ξ is a unit-variance first-order colored noise with a 120 s time constant

drawn on a fixed 10 s grid from the case's seed and interpolated, $\sigma = 0.05$, L ramps from 1 to $1+m$ over 10 min after a forecast miss of size $m = 0.25$, and F rises from 0 to 1 in 2 min, holds 6 min and falls over 4 min after a crowd of multiple $X = 4$ starts at t_X. The step uses the traffic at its midpoint, $t + \Delta t/2$, so that a finer step sees the same traffic.

14.3.2 One Step of the Request Fluid

Requests are tracked by attempt generation $g \in \{0,1,2\}$, the users' first attempts and their first and second retries, with arrival rates a_g and backlogs q_g. Over one step, with the active capacity $C = \mu n_{act}$, the balancer and the servers act as shown in Equation (14.2).

$$\begin{aligned} s &= \begin{cases} \min\big(1,\ c_{adm} C / \textstyle\sum a_g\big) & \text{admission on} \\ 1 & \text{off} \end{cases} \\ A_g &= s\,a_g, \qquad \text{shed}_g = (a_g - A_g)\,\Delta t \\ S &= \min\big(\textstyle\sum_g (q_g + A_g \Delta t),\ C\Delta t\big), \quad S_g \propto q_g + A_g \Delta t \\ w_g &= q_g + A_g \Delta t - S_g, \qquad q_g^{+} = w_g \min\Big(1,\ \frac{C T_q}{\sum w_g}\Big) \end{aligned} \tag{14.2}$$

where A_g is the admitted rate, S the requests served in the step, shared in proportion to what waits, and $w_g - q_g^{+}$ the overflow that the balancer refuses at the queue limit. With $\lambda_{off} = \sum_g a_g$ and $\lambda_{adm} = \sum_g A_g$, the step is exact for flows held constant over it, so the integration has no truncation error beyond the sampling of the traffic.

14.3.3 Latency: Flat, Then the Knee

A server runs c workers, each serving one job of T_{svc} at a time, so its capacity is $\mu = cB/T_{svc}$ with $B = 1$ for a web server. The latency of the step is the service time, the steady-state queueing wait at the present utilization and the fluid backlog drained at the fleet's rate, as shown in Equation (14.3).

$$\begin{aligned} \rho &= \frac{\lambda_{adm}}{\mu\, n_{act}}, \qquad U(\rho, c) = \frac{\rho^{\sqrt{2(c+1)}-1}}{c\,(1-\rho)} \\ W &= T_{svc}\big(1 + V\,U(\min(\rho, 0.98), c)\big) + \frac{\bar{q}}{\mu\, n_{act}} \\ p_{99} &= c_{99}\, W, \qquad f_{out} = 100^{-T_{out}/p_{99}} \end{aligned} \tag{14.3}$$

where $V = (c_a^2 + c_s^2)/2$ is the variability of arrivals and service, $\bar{q}$ is the mean backlog over the step, c_{99} is a declared ratio of the 99th percentile to the mean, and f_{out} is the share of served requests whose latency exceeds the client timeout under a tail declared exponential through p_{99}. At $c = 1$, U is Kingman's $\rho/(1-\rho)$ [7], and the exponent is Sakasegawa's multi-server form [15]. The term is held at $\rho = 0.98$ because above it the backlog carries the overload. With $c = 4$, $T_{svc} = 40$ ms, $V = 1$ and $c_{99} = 3$, the unloaded p_{99} is 120 ms, and the p_{99} is 133, 166, 213, 261, 359 and 657 ms at 50, 70, 80, 85, 90 and 95 % utilization, as shown in Figure 14.1(b). Queueing doubles the unloaded latency at 83.2 %, which the application calls the knee, and the 300 ms objective is crossed at 87.5 %. With one worker per server the knee would sit at 50 % and the objective at 60 %, so no target near 70 % could hold it, which is why the builder chose four.

The approximation is declared, so its error is stated. Against the exact M/M/4 mean wait from the Erlang C formula [8], at $V = 1$ where the two describe the same queue, the term is 28 % high at $\rho = 0.5$ (0.112 against 0.087 service times, a wait too small to matter), 3.5 % high at 0.8 and 1.1 % high at 0.9. The exact queue crosses the 300 ms objective at 87.7 % against the term's 87.5 %. The waiting term is the steady-state value at the present utilization, so a step in load reaches its new latency within the step instead of over the few seconds a real queue takes.

14.3.4 Timeouts, Retries and the Budget

Every refused, shed or timed-out request of generations 0 and 1 is sent again after T_{retry} with probability r, and a third failure is dropped, as shown in Equation (14.4).

$$\begin{aligned} F_g &= \text{shed}_g + \text{overflow}_g + f_{out} S_g \\ R_{g+1}(t + T_{retry}) &= r F_g k_\beta, \qquad g = 0, 1 \\ k_\beta &= \min\Big(1,\ \frac{\beta\, a_0 \Delta t}{r(F_0 + F_1)}\Big) \\ \text{dropped} &= (1 - r)(F_0 + F_1) + F_2 + \text{over budget} \end{aligned} \tag{14.4}$$

where R_{g+1} is the amount of generation $g+1$ that arrives one backoff later, $k_\beta = 1$ unless the retry budget is on, and $\beta = 10\,\%$ caps the retries scheduled in a step at a tenth of the users' requests in it, the figure of the overload chapter of the site-reliability book [9], as is the limit of three attempts. The default $r = 1$ retries every failure. Under the declared tail a 2 s timeout loses 0.01 % of the requests when p_{99} is 1 s, 1 % when it is 2 s and 10 % when it is 4 s.

The request ledgers close by construction, as shown in Equation (14.5).

$$\begin{aligned} \int \lambda_{user}\, \mathrm{d}t &= \text{good} + \text{dropped} + \textstyle\sum_g q_g + \text{pending retries} \\ \int \lambda_{off}\, \mathrm{d}t &= \text{served} + \text{shed} + \text{refused} + \textstyle\sum_g q_g \\ \text{cost} &= \int \left(n_{od}\, c_{od} + n_{spot}\, c_{spot}\right) \mathrm{d}t \end{aligned} \tag{14.5}$$

where good is served minus timed out, n_{od} and n_{spot} count every booting, active, draining and warm server billed at the on-demand and spot prices, and the scorecard's dropped share is dropped over the users' requests. On five disturbed runs, among them the challenge day with a preemption, a crowd and a storm, and case 2 with a slow dependency, a deploy, a forecast miss, a crowd and a 30 % spot preemption under the PI, every ledger closes to a relative 1.1×10^{-14} or better.

14.3.5 An Integer Fleet With a Dead Time

A server is an integer in a pipeline, ordered, booting for T_{boot}, active, and draining for T_{drain} out of rotation after a scale-in. A scale-in cancels booting servers first, the latest ordered first, and then drains active ones. A warm pool, off by default, keeps booted servers out of rotation at full price that join in $T_{warm} = 20\,\text{s}$ and is refilled by cold boots. A share of new servers may be bought on the spot market, and a preemption removes every spot server at once. A deploy restarts the active servers a quarter at a time. Every transition happens at the end of the first step that reaches its due time, so the boot delay is exactly 180 s (480 s for the model-serving fleet) from order to first service. Pumps with minimum run and rest times are integer actuators of the same kind in a water network (Volume 9, Chapter 11).

14.3.6 What the Platform Measures

The autoscaler reads only the platform's metric pipeline. Each 15 s slot accumulates the served work and the offered work in server-seconds, and at the end of each slot the averages over the last $T_m = 60\,\text{s}$ are published as the CPU busy fraction $u_{cpu} = \sum S / \mu \,/ \sum n_{act} \Delta t$, which cannot exceed one, and the requests per server against capacity $u_{req} = \sum \lambda_{off} \Delta t / \mu \,/ \sum n_{act} \Delta t$, which can. The minute's p_{99} is the served-weighted mean of the step values over each clock minute, and a minute above the objective is a breach minute. The counts of ready, booting and draining servers are known at once. The backlog, the true latency, the users' rate apart from their retries and the timeout share are truth columns, prefixed in the export and read only by the scene and the scorecard, and a test fails if the controller module reads any of them.

Table 14.2 Parameters. M: Show Model Dialog, C: Control Card, A: Advanced Control, S: Source. T: Teaching Value, Doc: Platform Documentation, SRE: Site-Reliability Book

Symbol	Meaning	Value	Unit		
λ_{peak}	evening peak of the profile	2400	req/s	M	T
σ, τ_ξ	traffic noise, its time constant	5, 120	%, s	C, S	T
X, m	crowd multiple, forecast miss	4, 25	—, %	A	T
T_{svc}, c	service time, workers per server	40, 4	ms, —	M	T
μ	capacity per server, c/T_{svc}	100	req/s	S	derived
V, c_{99}	variability, p_{99} over mean	1, 3	—	M	T
T_q	balancer queue limit	30	s	M	T
T_{out}, T_{retry}, r	timeout, backoff, share retried	2, 2, 1	s, s, —	M	T
—	attempts per request, budget β	3, 10	—, %	S	SRE
T_{boot}, T_{drain}	boot, drain	180, 30	s	M	T
n_{min}, n_{max}	floor, quota	2, 200	servers	M	T
c_{od}, c_{spot}	on-demand, spot price	0.20, 0.06	$/h	M	T
T_m	metric window	60	s	M	T
—	sync period, tolerance	15, 0.10	s, —	S, A	Doc
T_{cool}	scale-in stabilization window	300	s	C	Doc
—	scale-up limit per period, $\max(n, 4)$	—	servers	S	Doc
u_{SP}	utilization target	70	%	C	T
SLO	p_{99} objective	300	ms	C	T
c_{adm}	admission cap	85	%	C	T
K_c, τ_I, δ	PI gain, integral time, deadband	0.5, 300, 1	—, s, servers	C, A	T
Model-serving fleet, case 8					
c, B	GPUs per server, batch	2, 4	—	M	T
t_0, t_1	batch time $t_0 + t_1 B$	0.10, 0.05	s	M	T
T_{boot}, λ_{peak}	boot, peak	480, 300	s, req/s	M	T
—	SLO, target, cap, price	2, 60, 70, 2.50	s, %, %, $/h	C, M	T

14.3.7 Parameters and Numerics

Table 14.2 gives every parameter. All are teaching values except those read from a cited source, and none is a provider's price, instance type or trace.

The engine integrates in seconds at $\Delta t = 1$ s, the controller runs every 15 s, and the historian logs every 5 s. Every boot, drain, sync period, metric slot, backoff and minute is a whole number of steps. Halving the step to 0.5 s leaves the breach minutes and the reversals of every case unchanged, moves the dropped shares by at most 0.001 points and the costs by at most $0.01, except two day-long runs, the rule in the forecast's place ($90.12 against $90.28) and the PI ($98.80 against $98.92), and no challenge verdict changes. A simulated day costs 64 ms of computing headless and plays in about 24 s at the fastest speed in a browser. At the case 1 design point with the noise off the closed form gives $\rho = 0.6857$ and $p_{99} = 162.2$ ms, against 0.6854 and 162.2 ms from the engine.

14.3.8 Assumptions

The queue is a fluid, so requests are a continuous flow and the waiting time is the steady-state approximation at the present utilization, accurate to the figures above for $V = 1$. The percentile is a declared multiple of the mean with an exponential tail above it, while real services measure their own, and an exponential sojourn time would give $c_{99} = \ln 100 = 4.6$. There is one tier of identical servers behind a balancer that spreads load evenly, with no database, cache or downstream queue, so the slow-dependency event only lowers μ by 40 % for 10 min. A request whose client has timed out is still served, while servers that receive the client's deadline can drop it and shorten a storm. The rule is a written-out copy of the documented algorithm, not its code. Its readiness handling is reduced to booting servers counted at 0 % of the target on a scale-up and at 100 % on a scale-down, which the documentation applies to pods with missing metrics rather than to pods not yet ready. Billing is per second.

14.4 Autoscalers as Implemented

Every automatic mode decides every 15 s from the published metric u, which is u_{cpu} or u_{req} as the learner chooses, and the counts. Its result then passes the platform's two limits and the fleet is reconciled to it at once, which is the integer actuator, whose dead time is the boot.

14.4.1 The Platform's Rule

The rule computes a desired count as shown in Equation (14.6).

$$\begin{aligned} \phi &= u/u_{SP}, \qquad \text{no action if } |\phi - 1| \le 0.1 \\ \phi' &= \begin{cases} u\,n_{act}/n_{rep}\,/\,u_{SP} & \phi > 1 \\ (u\,n_{act} + u_{SP}\,n_{boot})/n_{rep}\,/\,u_{SP} & \phi < 1 \end{cases} \\ n_{des} &= \lceil \phi'\, n_{rep} \rceil \quad \text{unless } \phi' \text{ reverses or } |\phi' - 1| \le 0.1 \end{aligned} \tag{14.6}$$

where the recount ϕ' is made only while servers are booting. The copy reproduces the documented computation on declared inputs. Twenty ready servers at 90 % against a 70 % target give $\lceil 25.71 \rceil = 26$, a reading of 76 % is within the tolerance, twenty ready and four booting at 90 % recount to 1.071 and take no action, at 100 % they give $\lceil 1.1905 \times 24 \rceil = 29$, and at 40 % they give 16. A scale-in then takes the highest recommendation of the last T_{cool} seconds, a scale-out is limited to $n_{rep} + \max(n_{rep}, 4)$ per period, and the result is clipped to the floor and the quota.

As a controller the rule is proportional to the measured load, because $n_{rep}u/u_{SP}$ is the load in units of servers at the target. It has a deadband (the tolerance), a hold (the window), an integer output, a measurement lag of about half the 60 s window plus the publication delay, and a dead time equal to the boot. Counting booting servers as idle on a scale-up is supply-line accounting, so the rule does not order twice for load that servers on order will carry. On the CPU metric a saturated reading limits each decision to a multiple $1/u_{SP}$ of the ready fleet, which is $1/0.7 = 1.43$ at the shipped target.

> **+ DID YOU KNOW? A Cooling Time of at Least One Boot**
>
> A common practice against oscillation is a fixed minimum wait between scaling operations, at least as long as it takes to acquire, boot and configure a machine, and a long wait brings more violations because scale-out slows [3]. The rule of this chapter holds only scale-in, for 300 s against a 180 s boot (Table 14.2).

14.4.2 A Proportional-Integral Autoscaler

The PI option runs the shared velocity-form PID of Section 1.6.2 on the projected utilization $u\,n_{act}/n_{rep}$, the reading with booting servers counted as idle. It is direct acting, so a utilization above target raises the count, and its gain is scheduled so that $K_c = 1$ makes the proportional step equal to the rule's, as shown in Equation (14.7).

$$\begin{aligned} e_k &= \frac{u_k\, n_{act}}{n_{rep}} - u_{SP}, \qquad K_k = \frac{K_c\, n_{rep}}{u_{SP}} \\ s_k &= s_{k-1} + K_k e_k - K_{k-1} e_{k-1} \\ \hat{n}_k &= s_k + I_{k-1}, \qquad n_k^{PI} = \min\bigl(n_{max}, \max(n_{min}, \hat{n}_k)\bigr) \\ I_k &= I_{k-1} + \left[\frac{K_k}{\tau_I} e_k + \frac{n_k^{PI} - \hat{n}_k}{\tau_I}\right]\Delta t_c + \frac{n_k^{app} - n_k^{rec}}{\tau_I}\Delta t_c \end{aligned} \tag{14.7}$$

where $\Delta t_c = 15$ s, the running sum s_k is never clamped, and s_0 and I_0 are set so that the first output equals the starting fleet. The real-valued output becomes an order only when it leaves a deadband of δ servers around the present count (the integer deadband of Section 1.4.8), and the order is then $n^{rec} = \text{round}(n^{PI})$, at least one server away. The last term is the anti-windup against the limits that act after the PI, back-calculation with tracking time τ_I toward the count n^{app} that the cooldown and the

scale-up limit actually applied. The deadband is the PI's own and is not fed back. With no integral action nothing is back-calculated, and a PI that is not driving tracks the applied count for a bumpless switch.

The last term was added in the technical review. Without it, in case 2 at an 80 % target, the PI's order differed from the applied count on 3060 of the 14 400 steps, held by the cooldown or the scale-up limit, and its integral ran on through the hold. When the window let go, the PI removed a server on the rising ramp and the service fell into a retry storm, with 100 breach minutes, 22.3 % dropped and $18.50 over the four hours. Back-calculation against the applied count gives 70 breach minutes, no drops and $14.50 on the same run. Freezing the integral during a hold was also tried and rejected, because it stopped the scale-ins after the crowd and raised the challenge bill under the PI from $90.86 to $111.40.

14.4.3 The Forecast as a Floor

The forecast mode reads the declared diurnal profile at the planned peak, times one plus an adjustable bias, one lead ahead, and orders at least what that traffic needs at the target, as shown in Equation (14.8).

$$\begin{aligned} n_{ff} &= \frac{\lambda_{fc}(t+T_{lead})}{\mu\, u_{SP}}, \qquad n_{des} = \max\left(\lceil n_{ff} \rceil,\, n_{des}^{rule}\right) \\ T_{lead} &= T_{boot} + 15\,\mathrm{s} + T_m/2 = 225\,\mathrm{s} \end{aligned} \tag{14.8}$$

so the servers for a ramp are ordered one boot ahead of it while the rule still answers whatever the forecast misses. The design plan for the app called for a feedforward with a PI trim, and that structure was measured and replaced. On case 6 at an 80 % target with the miss, on five seeds, the floor breached 5, 24, 7, 9 and 12 minutes against 29, 80, 25, 35 and 34 for the trim at $K_c = 0.5$, and trims fast enough to catch the miss ($K_c = 0.8$) breached up to 123 minutes. In the idealized limit of a linear bill, a static utilization constraint and a known boot, the cost minimum orders the forecast's requirement one boot ahead, which the floor already computes. The predictive option below instead penalizes tracking error and count movement over a finite horizon.

14.4.4 Model Predictive Control

The model predictive controller (MPC) uses receding-horizon optimization [19] with tracking and movement penalties used in industrial MPC [20], set up as Section 1.6.3 describes. Nominal cold-provisioning physics gives the 12-state shift register of Equation (14.9).

$$\begin{aligned} x_{j,k+1} &= x_{j+1,k}, \quad j = 0,\ldots,10 \\ x_{11,k+1} &= v_k, \quad y_k = x_{0,k} \end{aligned} \tag{14.9}$$

where $x_{j,k}$ are pipeline counts, v_k is desired replicas, y_k predicted ready count and $T_s = 15\,\mathrm{s}$ the decision interval. Measured ready and booting counts and the controller's own orders initialize the register. Unknown boot ages are spread over 180 s, and newest orders cancel first. An isolated three-server cold addition has zero peak and RMS count residual over 600 one-second samples. Warm joins at 20 s, immediate scale-in and GPU boots at 480 s give RMS residuals of 1.549, 1.639 and 2.121 servers.

Demand is $d_k = u_k \max(1, n_{ready,k})/u_{SP}$, from the selected metric and measured ready count $n_{ready,k}$. Capacity stays at nominal $\mu_0 = 100\,\mathrm{req/s}$ per server. Additive bias compensation, related to offset-free MPC [21], supplies Equation (14.10).

$$\begin{aligned} f(t) &= \lambda_{fc}(t)/(\mu_0 u_{SP}), \quad a_k = 1 - e^{-\Delta t_k/(60\,\mathrm{s})} \\ b_k &= b_{k-1} + a_k\left(d_k - f(t_k) - b_{k-1}\right) \\ r_{i|k} &= \max\left(0, f(t_k + iT_s) + b_k\right), \quad i = 1,\ldots,40 \end{aligned} \tag{14.10}$$

where demand offset b_k starts at $d_0 - f(t_0)$ and Δt_k is elapsed decision time. Preview Off holds $r_{i|k} = d_k$. Forecasts use only the declared daily profile and adjustable bias, without future crowd, miss or noise. Windowed utilization times current ready count approximates demand during fleet changes. The crowd's fleet-to-window ratio reaches 1.25.

Shared LinearMPC/DenseQP solves eleven real counts z_ℓ over 40 steps (10 min), blocked as $(1,1,1,1,2,2,3,4,6,9,10)$, with objective J in Equation (14.11).

$$\begin{aligned} \min_{z} J = &\sum_{i=1}^{40} (y_{i|k} - r_{i|k})^2 + 0.20 \sum_{\ell=1}^{11} (z_\ell - z_{\ell-1})^2 \\ &+ \frac{0.20 T_s}{3600\,\mathrm{s}} \sum_{i=0}^{39} v_{i|k} \\ \text{subject to } & n_{min} \le z_\ell \le n_{max}, \quad |z_\ell - z_{\ell-1}| \le h_k \end{aligned} \tag{14.11}$$

where $v_{i|k}$ holds its block count, $z_0 = n_{rep,k}$ and $h_k = \min(h_{set}, \max(4, \lceil n_{rep,k} \rceil))$, default $h_{set} = 4$. Numerical counts are in servers and currency in dollars, at nominal price \$0.20 per server-hour. Box bounds (default 2–200) and symmetric block-boundary caps are hard. There is no terminal penalty, latency constraint or soft band, and slack is zero. Small default diagonal regularization applies. Only the first count is rounded, capped and clipped before stabilization, the platform scale-out policy and data-plane protections. Cooldown is not predicted. This is a relaxed QP with integer actuation.

Solves occur at initialization and every 15 s, holding commands between decisions. PI tracks the applied count. Nonoptimal, nonfinite or out-of-bounds plans hold that count on first failure, then use PI until recovery. Manual transfer retains the post-stabilization request before actuator caps. The nonlinear plant, instrument packet and frozen predictor stay separate. Backlog and hidden transition times are unavailable. Displayed p_{99} uses the nominal steady web curve without backlog.

14.4.5 Admission Control and the Retry Budget

Admission control is the override. With it on, the balancer admits at most $c_{adm} = 85\,\%$ of what the active servers can serve and refuses the rest at once as a fast error, so the utilization of the admitted flow cannot exceed 0.85 and p_{99} cannot exceed 261 ms once the backlog has drained. The refused requests are failures, so with $r = 1$ they come back as retries. A ramp signal that meters cars onto a freeway short of its breakdown is the same kind of override on another queue (Volume 2, Chapter 8), and Volume 4, Chapter 2 treats a protective override that rests at its clamp until the plant nears a limit. The retry budget acts on the clients instead, as Equation (14.4) states. Neither is a controller of the server count. A manual station sets the count directly, and a helper sets it to what the measured request rate needs at the target. Three models are kept apart. The plant is everything in Section 14.3, the rule carries no model beyond the counts, and the forecast knows only the declared profile and the capacity per server, not the crowd, the noise or the miss.

14.5 Results

The cases are listed in Table 14.3. Each starts from the shipped settings of Table 14.2 (the rule at 70 % on the CPU metric, a 300 s cooldown, admission control and the retry budget off, seed 20261026) at the clock time its row gives, with the fleet sized for the opening traffic at the target, and changes only what the row lists. The figures below come from headless runs of the engine, advanced step by step the way the browser advances it, over the window each sentence gives, and a setting that a sentence mentions was made before the run began. A case repeats bit for bit on its seed, as Section 1.6.1 explains. Table 14.4 collects the comparisons.

14.5.1 The Knee and the Boot by Hand

Case 1 opens at 15:00 in Manual with 28 servers, about 1920 req/s on average over the first ten minutes, a CPU reading averaging 69 % and a minute p_{99} of 151 to 192 ms. Removing servers at

Table 14.3 The Cases and the Challenge

#	Start, change and events	Teaching point
1	15:00, Manual at 28 servers	the knee and the boot
2	06:00, rule through the morning ramp	proportional, band, hold
3	21:30, cooldown 0 s, noise 8 %	flapping
4	19:45, crowd ×4 at 20:00	capacity after the crowd
5	case 4 with admission control	the override
6	06:00, forecast at 80 %, miss +25 % at 08:00	feedforward and its error
7	case 4 with a 60-server quota	the retry storm
8	19:00, GPU fleet, boot 480 s, crowd at 20:00	infeasible without shedding
9	06:00, MPC, Requests, target 70 %	ten-minute plan against the forecast floor
ch	00:00, 30 % spot, preemption at 14:00, crowd at 20:00	the whole day

Table 14.4 Measured Cases. Breach Minutes, Dropped Share of the Users' Requests and Bill Over the Window

Case, window	Setting	Breach (min)	Dropped (%)	Bill ($)
2, 06–10	rule 70 %	0	0	17.42
	rule 80 %	18	0	15.41
	PI 70 %	0	0	16.66
	PI 80 %	70	0	14.50
3, 21:30–01	cooldown 0 s	4	0	15.88
	cooldown 300 s	0	0	16.80
4, 19:45–21:45	as shipped	14	23.7	20.37
5, same	admission 85 %	0	14.0	15.81
	admission, Requests metric	0	4.8	21.37
6, 06–10	forecast 80 %	5	0	17.43
	rule 80 %	26	0	17.49
7, 19:45–23:45	as shipped	130	52.2	36.39
	retry budget	13	15.0	27.19
	admission control	0	8.7	26.91
8, 19–21	as shipped	18	27.4	136.28
	admission 70 %	0	16.4	101.51

15:10 moves along the knee. At 26 servers the p_{99} at 15:25 is 192 ms, at 24 it is 237 ms with no breach, at 23 the reading is 86.4 % and one minute breaches at 315 ms, and at 22 the reading is 90.3 % and five minutes breach, the worst at 466 ms. One server separates a clean quarter hour from a breaching one. Restoring 28 servers at 15:25 takes effect 180 s later, and the first clean minute ends at 15:29, three more breach minutes after the order. A removal acts in the next step and an addition one boot later, which is the asymmetry every autoscaler has to reason about.

14.5.2 The Rule on the Morning Ramp, and the PI

From 06:00 the traffic rises from 624 to 1920 req/s by 09:00. At a 70 % target the rule's reading averages 70.0 % from 07:00 to 09:00 with a peak of 80.7 %, and no minute breaches over the four hours, for $17.42. At 80 % the average is 79.6 %, the peak 91.8 %, and 18 minutes breach for $15.41. The rule chases the ramp from behind, because each decision acts on a reading about half a minute old and serves three minutes later, and the higher target leaves the slope of the knee less room for that lag. A saving of $2.01 buys 18 breach minutes.

The PI at its defaults ($K_c = 0.5$, $\tau_I = 300$ s) holds the 70 % ramp without a breach for $16.66 in 18 actions with none reversed, and at $K_c = 1$ it reverses 8 of 27 actions. At 80 % it is the cheapest, $14.50, and breaches 70 minutes, because a proportional controller with integral action follows a ramp with a standing error and its reading averages 85.3 % from 07:00 to 09:00 against the rule's 79.6 %. On the Requests metric the same PI gives the same 70 minutes, so the saturating CPU metric is not the cause. A target is a utilization, but the objective sits on the steep part of a curve, and a controller's error on a ramp has to be read on that curve.

🕮 **Pause.** Try Exercise 14.2 before reading on.

14.5.3 Flapping and the Cooldown

Case 3 runs the rule from 21:30 to 01:00 on the falling evening shoulder with 8 % noise and no cooldown. It takes 67 actions and reverses 29 of them, removing servers on quiet readings and ordering them back on the next burst at three minutes' notice, and 4 minutes breach for $15.88. A 60 s cooldown still reverses 19 times. The default 300 s takes 18 actions with none reversed and no breach, for $16.80. The hold costs $0.92 and removes the breaches, the reverse of the design plan's expectation that it would cost a breach, because here the reversals themselves caused them. On seeds 7 and 11 the counts without a cooldown are 37 and 28 reversals against 4 and 5 with it. Flapping is a limit cycle of a deadband controller on a noisy measurement with a delayed actuator, and the window is a hold that trades money for calm, the minimum dwell of Section 1.4.7 applied to scale-in only.

> **TRY IT IN THE APP** **3 · Flapping on the Evening Shoulder**
>
> Load the scenario and press Run. Watch the servers on the Day view and the Scaling actions tile. With no cooldown the rule removes servers on quiet readings and orders them back on the next burst. By 01:00 it has taken 67 actions and reversed 29 of them, and 4 minutes have breached.

14.5.4 The Flash Crowd and Admission Control

Case 4 is the crowd of the Introduction. From 36 servers at 20:00, the rule on the CPU metric can at most multiply the ready fleet by $1/0.7$ per boot, because the saturated reading cannot ask for more. The first new server serves at 20:03:45, 108 are active when the crowd ends at 20:12, and the fleet peaks at its 200-server quota at 20:16, sixteen minutes after the crowd began. The breach minutes end at 20:14. A boot of 20 s still leaves 12 breach minutes, because the multiplier per decision and the full queue's stale work matter as much as the boot. The state at 20:06 is shown in Figure 14.2 and listed in Table 14.1. Clients send 9614 requests per second and retries 18 715 more, the balancer refuses 23 129 at the queue limit, the backlog is 30 s of the 52 active servers' capacity, and of the 5200 req/s served only 498 arrive in time.

Pause. Try Exercise 14.1 before reading on.

Case 5 is the same crowd on the same seed with admission control on. The balancer admits at most 85 % of the active capacity and refuses the rest at once, so no minute breaches and the worst minute is 261 ms. Of the users' requests from 20:00 to 20:15, 51.1 % are answered in time against 17.2 % without it, and 14.0 % of the whole window's requests are dropped against 23.7 %, for $15.81 against $20.37. The design plan expected shedding five percent to save the rest, which does not hold for a fourfold crowd. The override also hides the overload from the CPU metric, which reads only what was admitted, so the fleet peaks at 73 servers. On the Requests metric, which counts the refused demand, the fleet reaches its quota in 4.8 minutes, 83.3 % of the crowd's requests are answered in time and 4.8 % are dropped, for $21.37. On seeds 7 and 11 the shipped crowd breaches 14 minutes each and the admitted one none.

> **DID YOU KNOW?** **Refusing Work Drops Less of It**
>
> From the simulator: in the fourfold crowd the shipped rule breaches 14 minutes and drops 23.7 % of the users' requests for $20.37. Admission control at 85 % of the active capacity breaches none, drops 14.0 % and bills $15.81 (Table 14.4, cases 4 and 5).

14.5.5 The Forecast

Case 6 runs the forecast floor at an 80 % target from 06:00, with the traffic climbing 25 % above the forecast from 08:00. No minute breaches before 08:00 against the rule's 17 at the same target, and five breach in all against 26, for $17.43 against $17.49. Without the miss the forecast breaches 1 minute against 18. The floor orders the ramp a lead of 225 s ahead, and the rule above it answers the miss.

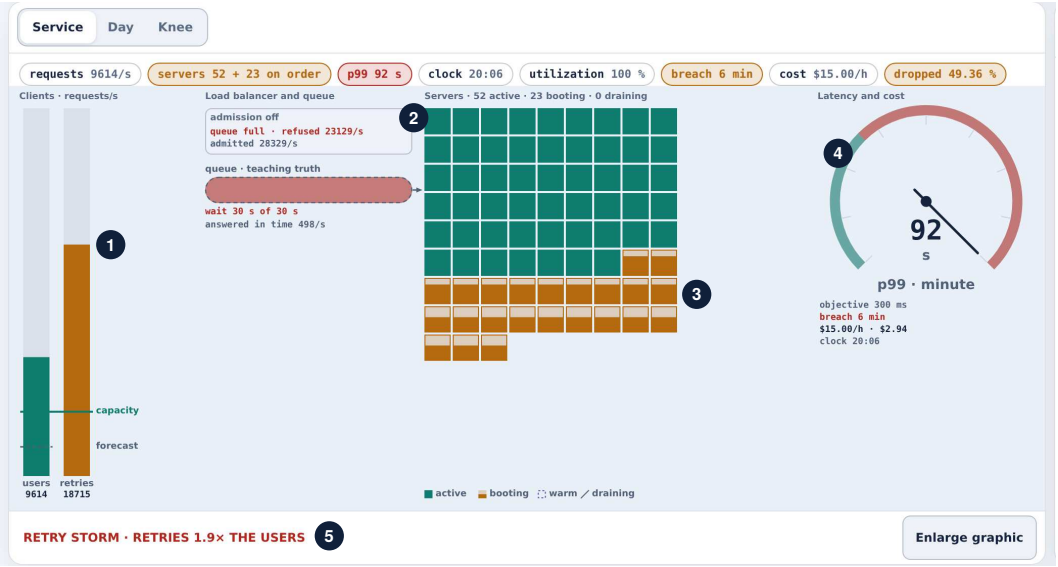


Figure 14.2 The flash crowd of case 4 at 20:06 on the Service view, as shipped (rule at 70 % on the CPU metric, admission control off). The picture is cropped to the Service view, and the header, the control panel, the trends and the Learn panel are cut. (1) The clients' bars show 9614 users and 18 715 retries per second. (2) The balancer refuses 23 129 per second at its queue limit, the queue drawn as teaching truth is full at 30 s of the capacity, and 498 requests per second are answered in time. (3) The rack holds 52 active and 23 booting servers. (4) The gauge reads a minute p_{99} of 92 s against the 300 ms objective and a bill of $2.94 so far. (5) The status strip reads RETRY STORM at 1.9 retries per user. The chip row above the scene reads the utilization pinned at 100 %, 6 breach minutes and 49.36 % of the users' requests dropped since 19:45.

14.5.6 The Retry Storm

Case 7 is the crowd of case 4 with the quota cut to 60 servers, a fleet that can serve 6000 req/s against an evening peak of 2400 req/s. Without a budget, the retries exceed the users in every minute from 20:02 to 22:10, 129 minutes, and 130 minutes breach. The trigger lasted twelve minutes. At 21:00 the users send 2629 requests per second, 44 % of the capacity, yet the minute p_{99} is 91.6 s, because every request now arrives up to three times and the retries run at 1.64 to 1.78 times the users. The service has two states at the same traffic. Healthy, almost nothing fails and the offered load is the users' load. Overloaded, almost everything fails, the offered load is near three times the users' load, the queue stays at its limit, and the overload sustains itself. The loop is a positive feedback, failures cause retries and retries cause failures. This one ends only when the evening traffic has fallen far enough that users and retries together stay below what the fleet serves for long enough to drain the full queue, which happens at 22:10.

🕮 **Pause.** Try Exercise 14.3 before reading on.

The budget breaks the loop. Table 14.5 pairs the shipped case with the same case under the 10 % budget on ten seeds. Without it, nine seeds storm for 107 to 129 minutes and one, seed 3, for 22 minutes, because there the evening noise dipped the users low enough to drain the queue at 20:23. With the budget, no seed has a minute with more retries than users, the largest ratio is 0.10, and the breach minutes fall to 12 or 13, the crowd's own. A sweep of the share of failures retried on the shipped seed shows the gain of the loop. At $r = 0.5$ there is no storm minute, at 0.6 one, at 0.7 fourteen, at 0.8 seventeen, at 0.9 twenty-eight and at 1.0 the full 129. A crowd of 1.5 times the traffic breaches 4 minutes with no storm, while 2, 2.5 and 3 times all storm to 22:10. Admission

Table 14.5 Case 7 on Paired Seeds, 19:45 to 23:45. Storm Minutes Have More Retries Than Users

	No budget			Budget 10 %		
Seed	storm (min)	breach (min)	dropped (%)	storm (min)	breach (min)	dropped (%)
20261026	129	130	52.2	0	13	15.0
1	114	115	46.9	0	13	14.8
2	118	120	48.6	0	12	14.6
3	22	24	16.2	0	12	13.2
4	118	120	49.0	0	12	15.4
5	107	109	45.1	0	13	15.3
7	127	128	51.1	0	12	15.4
11	115	117	48.4	0	13	15.5
23	121	122	48.7	0	12	14.7
99	121	123	49.8	0	12	15.2

control alone also prevents the storm's breaches (none), because a fast refusal keeps the latency of the admitted requests under the objective, although it still drops 8.7 % of the requests and retries still exceed the users for nine minutes.

> **TRY IT IN THE APP 7 · The Retry Storm**
>
> Load the scenario and press Run. From 20:02 the retries outnumber the users, and the service stays in overload until 22:10, two hours after the crowd ended. Switch Retry budget (10 %) on, press Reset and run again: no minute has more retries than users, and 13 minutes breach in all.

14.5.7 A Fleet That Cannot Be Saved Without Shedding

Case 8 serves a model on GPU servers, two GPUs each, that batch four requests in $t_0 + t_1 B = 0.3$ s, so a server carries 26.7 req/s, the unloaded p_{99} is 0.9 s against a 2 s objective, and a server boots in 480 s. Batching trades latency for throughput. At the objective a server answers 16.7, 19.7 and 15.4 req/s in time at batches of 2, 4 and 8, and a batch of 8 breaches 120 minutes of the two hours even without a crowd. With the crowd at 20:00 the first new server serves at 20:08:30, as the crowd begins to fall, 18 minutes breach and 27.4 % of the requests are dropped. A sweep of 168 configurations without admission control (rule, PI and forecast, both metrics, targets from 30 to 90 %, warm pools of 0, 5, 10 and 20 servers) finds none within 5 breach minutes. The best, the rule at 30 % with 20 warm servers, breaches 7 minutes for $387.34 in two hours. With admission control at 70 % the shipped autoscaler breaches none and drops 16.4 %, for $101.51. The objective is infeasible through the crowd for all 168 tested configurations, and the case says so. Only shedding keeps it.

14.5.8 The Day

The graded day starts at midnight and lasts 24 hours, with 30 % of new servers bought spot, a preemption at 14:00 and the crowd at 20:00. Its target is at most 5 breach minutes, at most 1 % of the users' requests dropped and a bill of at most $92. The drop target is 1 % rather than the design plan's 0.5 % because no autoscaler can do better than about 0.6 % once the crowd has begun. An oracle that, under the suggested settings, switches to Manual and orders 130 servers at 20:00:00 for twenty minutes still drops 0.599 % of the day's requests, and 160 servers drop 0.595 %, against 0.038 % for the same day without the crowd. Summed over the first three minutes of the crowd, the users' requests above what the starting fleet can admit come to 0.80 million, 0.57 % of the day's 139.7 million. Table 14.6 compares nine configurations.

The shipped rule fails all three clauses, with 3 breach minutes at the 14:00 preemption and 14 in the crowd, 3.49 % dropped and $94.21. The suggested configuration, the forecast at 75 % on the Requests metric with admission control, passes at 0 breach minutes, 0.82 % and $89.49. What does the work there is the metric and the override, not the forecast. With admission control on, a ramp

Table 14.6 The Day in Nine Configurations (Target: ≤ 5 Breach Minutes, ≤ 1 % Dropped, ≤ $92)

Configuration	Breach (min)	Dropped (%)	Bill ($)	Verdict
shipped: rule 70 %, CPU	17	3.49	94.21	fails
rule 70 % + admission	0	2.01	90.15	fails
PI 70 %, Requests, admission	0	0.87	98.80	fails
forecast 70 %, Requests, admission	0	0.78	95.86	fails
forecast 75 %, Requests, admission (suggested)	0	0.82	89.49	passes
suggested on CPU	0	2.31	84.25	fails
suggested with the rule in place of the forecast	0	0.78	90.12	passes
suggested + retry budget	0	0.95	88.85	passes
suggested + two warm servers	0	0.76	103.59	fails

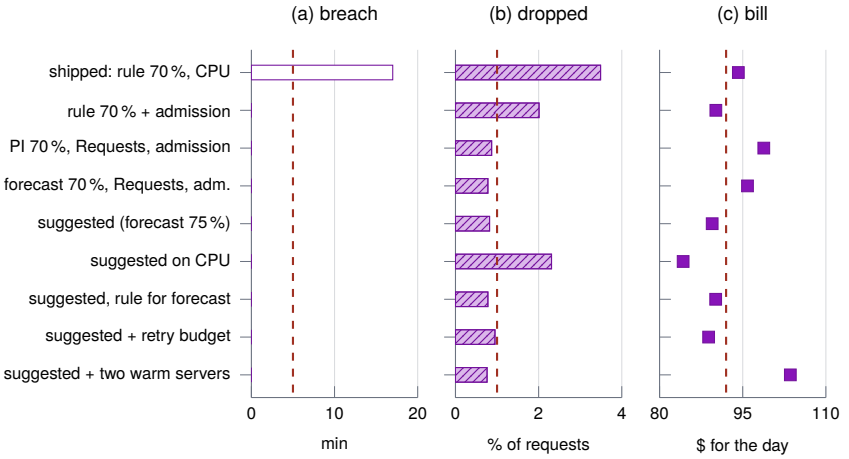


Figure 14.3 The graded day of Table 14.6 in its nine configurations: (a) breach minutes as white bars, (b) the share of the users' requests dropped as shaded bars and (c) the bill as dark squares on an axis that starts at $80, with each target dashed. The table prints the values. Notice that every configuration with admission control keeps the breach target, and that the drop and bill targets then separate them: only the rows on the Requests metric drop less than 1 %, and the warm servers that drop least cost the most.

cannot breach, because the admitted utilization is capped at the 261 ms point, and the rule in the forecast's place also passes, at 0.78 % and $90.12, so the forecast's contribution on this day is $0.63 of the bill, at 0.04 points more of the requests dropped. The same pieces at the shipped 70 % fail on cost, $95.86, and on the CPU metric the override hides the crowd from the rule and 2.31 % is dropped. The warm pool the design plan suggested cuts the drops to 0.76 % but costs $14.10 more. On nine further seeds the shipped rule fails with 14 to 28 breach minutes and 3.46 to 4.54 % dropped, and the suggested configuration passes every time at 0.80 to 0.89 % and $89.78 to $90.92. Figure 14.3 draws the nine configurations against the three targets.

14.5.9 Predictive Orders Against the Forecast Floor

Case 9, "A ten-minute plan against the forecast floor," repeats 06:00–09:00 with Requests and a 70 % target. Table 14.7 matches seed 20261026, instruments, noise and case settings, with MPC weight 0.20 and cap 4 unchanged. Cases 6 and 4 retain targets of 80 and 70 %. Forecast and MPC On receive the same daily schedule without future crowd or miss events. Rule and PI have no preview. Off holds measured demand. The earlier CPU results remain separate.

All five ramp runs have zero breaches and no drops. PI is cheapest at $11.06, against Rule $11.81750, MPC On $11.93333 and Forecast $12.08333. Preview reduces the MPC bill from

Table 14.7 Matched Requests Comparisons, Seed 20261026. MPC Settings Stay Fixed

Case, window	Law	Breach (min)	Dropped (%)	Billed (server h)	Bill ($)	IAE (util·s)	Travel (servers)	Limit (s)	Solves/ fallback	Mean (µs)
9, 06–09	Rule	0	0.000	59.087	11.81750	412.9	22	0	0/0	—
	PI	0	0.000	55.300	11.06000	622.9	18	0	0/0	—
	Forecast	0	0.000	60.417	12.08333	341.9	20	0	0/0	—
	MPC Off	0	0.000	60.233	12.04667	336.6	36	0	720/0	175.9
	MPC On	0	0.000	59.667	11.93333	303.2	27	0	720/0	182.4
6, 06–10	Rule	26	0.000	87.438	17.48750	718.0	33	0	0/0	—
	PI	97	7.262	106.288	21.25750	2095.2	91	0	0/0	—
	Forecast	5	0.000	87.171	17.43417	496.1	21	0	0/0	—
	MPC Off	8	0.000	89.246	17.84917	546.4	51	0	960/0	174.8
	MPC On	11	0.000	88.400	17.68000	510.0	42	0	960/0	181.3
4, 19:45–21:45	Rule	12	19.671	118.292	23.65833	1842.6	327	900	0/0	—
	PI	12	20.444	152.096	30.41917	3020.6	336	720	0/0	—
	Forecast	12	19.671	118.292	23.65833	1842.6	327	900	0/0	—
	MPC Off	13	22.368	289.200	57.84000	5222.6	242	360	480/0	198.3
	MPC On	13	22.492	289.437	57.88750	5319.9	242	390	480/0	199.2

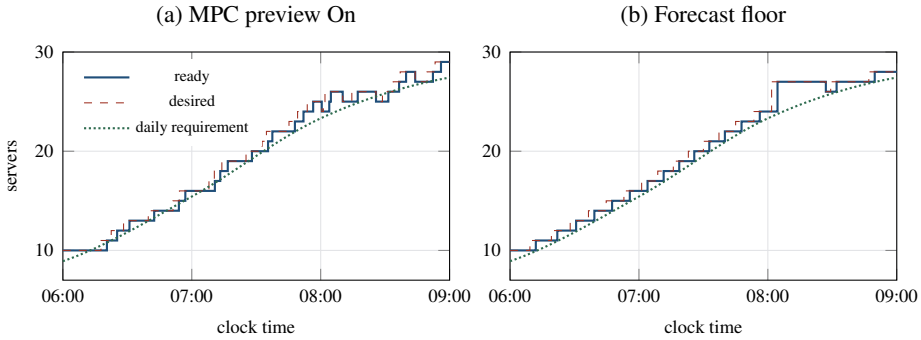


Figure 14.4 Morning server orders under MPC (a) and Forecast (b). Both use Requests, a 70 % target and seed 20261026 through 09:00. Ready counts (solid) and desired orders (dashed) are compared with the daily forecast's capacity requirement (dotted). Neither run breaches, at bills of $11.93333 and $12.08333, while PI costs $11.06 on the same ramp.

$12.04667 to $11.93333, while its IAE falls from 336.6 to 303.2 utilization-seconds. The optimized and floor-based order trajectories differ despite identical zero-breach outcomes, as shown in Figure 14.4. Removing only the linear price term changes one of 720 rounded ramp decisions and adds $0.000833 to the bill, so a material price-optimization benefit is not supported.

MPC loses both disturbed comparisons. On the forecast miss, On breaches 11 minutes for $17.68 against Forecast's 5 for $17.43417, and Off's 8 for $17.84917. Its applied count falls below the forecast's 225 s floor at 164 of 960 decisions. The move cap never binds there. On the crowd, Rule and Forecast give 12 breach minutes and 19.671 % drops for $23.65833, against MPC On's 13, 22.492 % and $57.88750. Its cap binds at 410 of 480 decisions. The count cap and absence of a forecast floor are distinct limitations. Neither observation establishes a unique cause of the losses.

With the unchanged suggested challenge settings, replacing Forecast by MPC gives zero breaches but 1.324 % drops and $121.17363, failing both drop and cost clauses. Forecast passes at zero, 0.816 % and $89.48888. No tuning or challenge requirement is changed. Hours integrate billed on-demand and spot counts before transitions. IAE integrates absolute metric error every 5 s, travel sums applied count changes and limit time counts either count bound. Table timings include QP preparation, excluding initialization. Chrome measured 2900 µs over three startup samples at real

time and 172.1 µs over 2884 warm samples at 3600×, with zero fallbacks. Host and warm-up affect timing.

14.6 Verification, Agreement and Evaluation

Three kinds of evidence support the application, and they are kept apart. Code verification covers the ledgers of Equation (14.5), which close to 1.1×10^{-14} relative, the bill against an independent sum of billed servers over the day to 10^{-12}, the boot delay of exactly 180 and 480 s, the rule against the documentation's computations on six declared inputs, Kingman's term at $c = 1$ to 10^{-12} at three utilizations, every scripted event marked once, and the step halving above. The application now runs 53 checks, including the predictive controller. Agreement with published results is limited to the queueing law, where the declared term is within 3.5 % of the exact M/M/4 wait from 80 % utilization upward. The retry amplification under the budget, a largest ratio of 0.10, is the source's figure by construction [9], not an independent agreement. No traffic trace, price or fleet of a real service was fitted, so no agreement with operating data is claimed. Educational evaluation is limited to play tests made while the app was reviewed, in which a first breach appeared after 23 s of wall time in case 1 and the challenge was passed from its hints in three runs, about 1.5 min. No learning study has been made.

14.7 Technology in Practice

What Measures It

The autoscaler of a real service reads a metric, not a server. On the open-source container platform whose rule this chapter copies, the resource metrics pipeline offers only the minimum CPU and memory metrics needed for automatic scaling, and a fuller set needs a second pipeline that serves custom metrics [22]. The autoscaler can scale on such custom metrics, or on an external metric fetched as a single value [14]. The CPU busy fraction is the chapter's u_{cpu}, and a request rate per server is its u_{req}, the Requests option that in case 5 counts the refused demand the CPU reading hides.

> **TECHNOLOGY Sensor: A Metrics Pipeline**
>
> On the open-source container platform the reading is the end of a pipeline. A node agent derives each container's CPU use as a rate over a cumulative counter kept by the kernel, over a window it reports with the value, and a cluster add-on collects these readings every 15 s for the autoscaler [22, 23]. The add-on's own documentation warns against using it as an accurate source of resource usage [23]. A starting server can read misleadingly high while it initializes, so by default the autoscaler ignores a new pod's CPU for its first five minutes unless the sample was taken while the pod was ready, and a pod with no metric is set aside and counted conservatively [14]. The chapter's u_{cpu}, a 60 s mean published every 15 s that cannot exceed one, models this pipeline.

What Moves It

Two autoscalers stand between an order and the capacity it buys. The horizontal autoscaler sets the replica count of a workload, and a cluster autoscaler adds machines when pods cannot be placed and removes machines that stay unneeded [24]. Its documentation discourages node autoscalers that act on CPU use alone, because they may add a node with no pods on it or remove one that carries system-critical pods [24]. The chapter merges the two layers into one fleet of servers that each boot in 180 s, and Figure 14.5 draws the loop on this hardware.

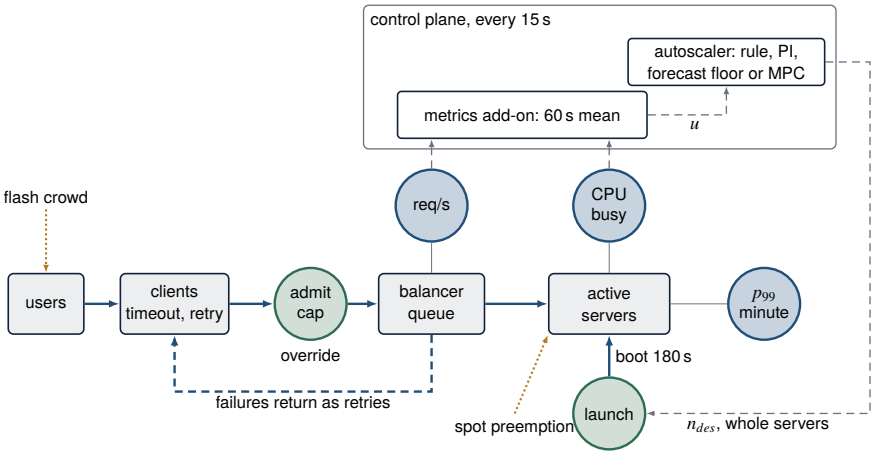


Figure 14.5 The autoscaling loop drawn on the hardware of an open-source container platform, with the chapter's symbols. Sensors are drawn blue, final elements green and disturbances dotted. The metrics add-on turns the CPU busy fraction or the request rate into the published u, the autoscaler orders whole servers, and each order becomes capacity only after a launch and a boot. Notice that the p_{99} the objective is set on is an input of no autoscaler, and that admission control acts at the balancer, outside the count loop.

> **TECHNOLOGY** **Final Element: A Machine Launch**
>
> A replica count is a command, and capacity arrives only when a machine exists to run it. A cluster autoscaler checks every 10 s for pods that the scheduler could not place and asks the provider for a node from a group of identical machines. How long the node takes to appear depends almost entirely on the provider and the speed of provisioning. The autoscaler expects it within 15 minutes, after which it stops counting on it, may try another group and removes a node that never registered. Removal is slow on purpose: a node must stay unneeded for 10 minutes, and non-empty nodes are removed one at a time [24]. The chapter folds all of this into one 180 s boot and a 30 s drain (Table 14.2).

Where the Controller Runs

The horizontal autoscaler is a control loop that runs intermittently inside the cluster's controller manager, by default every 15 s [14], which is the control period of every automatic mode in this chapter. Each autoscaler object holds the structure of the platform's rule: the target, a tolerance of 0.1 by default, a scale-down stabilization window of 300 s, and scaling policies that cap the change in one period [14]. The documentation names the problem the window treats, a replica count that keeps fluctuating with the metric, which it calls thrashing or flapping and likens to hysteresis [14]. The cluster autoscaler under it checks every 10 s [24].

> **IN PRACTICE** **Load Test to the Breaking Point**
>
> Site-reliability practice calls a load test of a server's capacity limits, and of its failure mode under overload, the most important exercise for preventing overload. Components are loaded until they break, both by a gradual ramp and by an impulse, because caching makes the two differ, and are then returned to nominal load to see whether they leave the degraded mode without help. The breaking point feeds capacity planning: at 5000 queries per second per cluster and a peak of 19 000, about six clusters run the service with two spare [9]. Case 1 walks the knee by hand, and case 7 shows a service that does not leave overload when its trigger ends.

What Is Modernizing It

Three developments touch this loop. Scale-up and scale-down behaviors are configured separately, each with its own stabilization window and policies, and a tolerance per direction has been stable since version 1.37 [14], so the hold of case 3 can be set for each direction. Scaling to and from zero replicas is now on by default as a beta feature, for object and external metrics only, because CPU can be measured only on running pods [14]. Warm capacity is kept with low-priority placeholder pods that hold resources until real pods preempt them, which forces the cluster autoscaler to add a node before it is needed [24]. That is the role of the warm pool in case 8 and in the last row of Table 14.6.

14.8 Limitations

The model is a fluid queue with a declared latency term, so it has no individual requests, no burst within a second and no service-time distribution beyond V, and its p_{99} is a multiple of the mean. Bursty traffic would move the knee to the left. With one tier and no downstream queue, a slow database appears only as a lower capacity. The rule's readiness handling is simplified, and the node autoscaler under the servers, placement across zones and startup bursts are absent. The retry storm depends on $r = 1$, on three attempts and on a quota that caps the fleet, and its length on how far the evening traffic falls, so its two hours belong to this case and are not a prediction for a real service. Every price, boot time and objective is a teaching value, the comparisons are seeded teaching results that say nothing about the general optimality of a policy, and a simulated pass is not evidence that a real service would meet a contractual service level.

> **OPEN CHALLENGE** **Predicting the Next Metastable Failure**
>
> Many production systems run in a vulnerable state for efficiency, and a systematic way to build systems robust against unknown metastable failures remains an open problem. Detecting a vulnerable state is difficult because of the sheer size of the systems, and predicting a failure is harder still, since it needs the state, the trigger and its intensity. The defenses in use are often reactions to earlier failures and are not applied systematically. One proposal is to watch a characteristic metric, such as queueing delay or the timeout rate, that the trigger disturbs and that recovers only after the failure ends [10]. In this chapter the share timing out is a truth column that no controller reads.

The MPC fixes web capacity and cold delay even when service time or boot settings change. It omits backlog, retries, immediate scale-in, warm joins, GPU dynamics, drain and warm bills, and spot prices. Measured counts re-anchor its prediction but do not remove these mismatches. Its finite-horizon count-tracking objective neither enforces the latency objective nor minimizes the measured bill.

> **OPEN CHALLENGE** **Forecasting the Crowd**
>
> Existing autoscalers mostly predict the workload from its own past. A survey proposes using social media and other real-time channels to foresee bursts, and notes that a general-purpose predictor of this kind is difficult to design [3]. Progress would be a forecast that sees a crowd one boot ahead. The forecast floor of this chapter knows only the declared daily profile, and in case 4 it breaches as often as the rule (Table 14.7).

14.9 Conclusion

This chapter has treated an autoscaled web service as a plant with an integer actuator, a boot dead time, a saturating metric, a latency knee and a positive feedback through its clients. Queueing doubles the unloaded latency at 83.2 % utilization and the 300 ms objective falls at 87.5 %, within 3.5 % of the exact queue near the knee. The published rule at a 70 % target follows a morning ramp without a breach, yet leaves 14 breach minutes in a fourfold crowd whose servers arrive after it has gone. A 300 s cooldown removes 29 reversals for $0.92. Admission control keeps the objective by refusing work, and on the Requests metric it drops 4.8 % of the crowd window's requests instead of 23.7 %. With the fleet capped, retries hold the service in overload for two hours at 44 % of its capacity, and a 10 % retry budget removes every storm minute on ten of ten seeds. On the graded day the metric and the override do the work, and the forecast saves $0.63. The predictive option costs $11.93 against PI's $11.06 on the three-hour ramp, and loses to the forecast on both disturbed comparisons. The model is a fluid with a declared percentile. Further work would add a downstream dependency with its own queue, request deadlines that cancel abandoned work, a measured traffic trace and a fitted tail. The simulator shows that the capacity a feedback loop orders must be judged against the time it takes to arrive.

KEY POINTS

- Servers ordered by feedback arrive one boot late, so a rule that rides the morning ramp leaves 14 breach minutes in a fourfold crowd.
- The rule is proportional to the measured load, with a deadband, a hold, an integer output and a dead time equal to the boot.
- A 300 s cooldown removes 29 reversals for $0.92.
- Admission control keeps the objective by refusing work.
- With the fleet capped, retries hold the service in overload for two hours, and a 10 % retry budget removes every storm minute on ten of ten seeds.

Where else this applies. The scale-in hold is a minimum dwell of the same kind as the minimum run and rest times of the rack compressors in Chapter 12 (Table 12.4).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[3] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: a taxonomy and survey," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–33, 2018.

[4] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. Wiley, 2004.

[5] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "AutoScale: dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems*, vol. 30, no. 4, pp. 1–26, 2012.

[6] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[7] J. F. C. Kingman, "The single server queue in heavy traffic," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 57, no. 4, pp. 902–904, 1961.

[8] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems*. Cambridge University Press, 2013.

[9] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering*. O'Reilly Media, 2016.

[10] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu, "Metastable failures in distributed systems," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, 2021, pp. 221–227.

[11] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for Web server end-systems: a control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, pp. 80–96, 2002.

[12] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, 2009, pp. 13–18.

[13] R. Isaacs, P. Alvaro, R. Majumdar, K.-K. Muniswamy-Reddy, M. Salamati, and S. Soudjani, "Analyzing metastable failures," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '25)*, 2025, pp. 172–178.

[14] The Kubernetes Authors, "Horizontal pod autoscaling," Kubernetes documentation, https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/, 2026, page last modified August 3, 2026; accessed October 26, 2026.

[15] H. Sakasegawa, "An approximation formula $L_q \simeq \alpha \cdot \rho^{\beta}/(1-\rho)$," *Annals of the Institute of Statistical Mathematics*, vol. 29, no. 1, pp. 67–75, 1977.

[16] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.

[17] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Gaithersburg, MD, Special Publication 800-145, 2011.

[18] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[19] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, no. 6, pp. 789–814, 2000.

[20] S. J. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," *Control Engineering Practice*, vol. 11, no. 7, pp. 733–764, 2003.

[21] G. Pannocchia and J. B. Rawlings, "Disturbance models for offset-free model-predictive control," *AIChE Journal*, vol. 49, no. 2, pp. 426–437, 2003.

[22] The Kubernetes Authors, "Resource metrics pipeline," Kubernetes documentation, https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/, 2026, page last modified July 1, 2026; accessed November 22, 2026.

[23] "Kubernetes metrics server," Project README, https://github.com/kubernetes-sigs/metrics-server, 2026, revision of November 2, 2026; accessed November 22, 2026.

[24] "Cluster autoscaler FAQ," Kubernetes autoscaler project, https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md, 2026, revision of September 26, 2026; accessed November 22, 2026.

14.10 Exercises

14.1 ○ Concept check · 2 min

In case 4 the traffic quadruples at 20:00 and the fleet reaches its 200-server quota only at 20:16, after the crowd has passed. What limits how fast the platform's rule on the CPU metric grows the fleet?

(a) The scale-in cooldown holds the fleet for 300 s after every order, so the orders are one window apart

(b) The reading is pinned at 100 %, so a decision asks for at most $1/0.7$ times the ready servers, and the next step waits for a boot

(c) The rule counts booting servers as busy, so it orders twice for the same load and then has to wait for the excess to drain

(d) The metric is a 60 s mean published every 15 s, so the rule cannot see the crowd until the crowd is over

14.2 ○ Concept check · 2 min

On the morning ramp of case 2 at an 80 % target, the PI at its defaults costs \$14.50 and breaches 70 minutes, while the rule costs \$15.41 and breaches 18. Why does the PI breach so much more?

(a) The CPU metric saturates, so the PI cannot see how far above the target the servers run

(b) The PI counts booting servers as busy, so it waits for every boot before it orders again

(c) A PI follows a ramp with a standing error, and its reading averages 85.3 %, close to the 87.5 % where the objective falls

(d) Back-calculation against the cooldown freezes the integral for the whole of the ramp

14.3 Concept check · 1 min

In case 7 at 21:00 the users send 2629 requests per second, 44 % of what the 60-server quota can serve, yet the minute p_{99} is 91.6 s. Which statement explains it?

(a) The 60-server quota leaves too little capacity for the evening users

(b) The boot delay keeps the servers ordered during the crowd from arriving

(c) Admission control refuses most requests, and every refusal is counted as a breach

(d) Failures cause retries and retries cause failures, so the offered load stays near three times the users' and the queue stays full

14.4 Theory and calculation · 6 min

Use Equation (14.3) with the shipped values $c = 4$, $T_{svc} = 40$ ms, $V = 1$ and $c_{99} = 3$, with no backlog.

(a) Find p_{99} at a utilization of 80 %. *Checkpoint:* 213 ms.

(b) Find p_{99} at 85 %, and say why admission control with an 85 % cap keeps every minute under the 300 ms objective once the backlog has drained. *Checkpoint:* 261 ms.

(c) With one worker per server ($c = 1$), find the utilization at which queueing doubles the unloaded latency and the one at which p_{99} reaches 300 ms. *Checkpoint:* 50 % and 60 %.

14.5 Theory and calculation · 7 min

The platform's rule (Equation (14.6)) runs at a 70 % target with the 10 % tolerance.

(a) Twenty ready servers read 90 % and none is booting. Find the desired count. *Checkpoint:* 26.

(b) Twenty ready servers and four booting read 90 %, then 100 %. Find the recount ϕ' and the action in each case. *Checkpoint:* 1.071, no action; 1.190, 29 servers.

(c) In case 4 the CPU reading is pinned at 100 % from 36 ready servers. List the counts the rule can reach, one boot at a time, until the 200-server quota, and estimate when the last step serves if the first full order goes out at 20:01. *Checkpoint:* 52, 75, 108, 155, 200; about 20:16.

14.6 Theory and calculation · 6 min

In case 7 the fleet is capped at 60 servers of 100 req/s, and at 21:00 the users send 2629 req/s. Assume that in the overloaded state almost every attempt fails, that a failure is retried with probability r and that a request makes at most three attempts.

(a) Find the offered load in the overloaded state at $r = 1$, and compare it with the capacity. *Checkpoint:* 7887 req/s against 6000 req/s.

(b) Find the largest offered load with the 10 % retry budget on, as a share of the capacity. *Checkpoint:* 2892 req/s, 48 %.

(c) Find the share r below which the overloaded state cannot hold itself at this traffic, even with every attempt failing. *Checkpoint:* 0.74.

14.7 App challenge · 8 min

Load Scenario 2, *The platform rule and the morning ramp*, at the default seed and change nothing. Run past 10:00 (4 h) at speed 600 (about 24 s), pause, export the CSV file and read columns `breach_min` and `cost_usd` in the row whose `clock_h` reads 10. Reload the scenario, set the Utilization target to 80 % before Run, and repeat. Reload once more, set the Autoscaler to PI and the target to 80 % before Run, and repeat.

(a) Give the breach minutes and the bill of the rule at 70 and at 80 %. *Checkpoint:* 0 and $17.42; 18 and $15.41.

(b) Give the same for the PI at 80 %, and the clock time of the last breach minute in both 80 % runs. *Checkpoint:* 70 and $14.50; 08:37.

(c) What does the rule's higher target buy, and what does it cost? *Checkpoint:* $2.01 for 18 breach minutes.

14.8 App challenge · 8 min

Load Scenario 4, *The flash crowd*, at the default seed and change nothing. Run past 21:45 (2 h) at speed 600 (about 12 s), pause, export the CSV file and read `breach_min`, `dropped_pct` and `cost_usd` in the row whose `clock_h` reads 21.75, and the largest value of `servers_active`. Repeat with Scenario 5, *Admission control: refuse at once, answer the rest*, as loaded, and again with Scenario 5 after setting the Scaling metric to Requests before Run.

(a) Give the four values for case 4. *Checkpoint:* 14, 23.7 %, $20.37, 200 servers.

(b) Give them for case 5 on the CPU metric. *Checkpoint:* 0, 14.0 %, $15.81, 73 servers.

(c) Give them for case 5 on the Requests metric, and explain why the fleet peaks so differently on the two metrics. *Checkpoint:* 0, 4.8 %, $21.37, 200 servers.

14.9 App challenge · 8 min

Load Scenario 7, *The retry storm*, at the default seed and change nothing. Run past 23:45 (4 h) at speed 600 (about 24 s), pause, read the Retries over users tile, export the CSV file and read `breach_min`, `dropped_pct` and `cost_usd` in the row whose `clock_h` reads 23.75. Reload the scenario, switch Retry budget (10 %) on before Run, and repeat.

(a) Give the minutes with more retries than users, the breach minutes, the dropped share and the bill without the budget, and the clock time at which `breach_min` last increases. *Checkpoint:* 129, 130, 52.2 %, $36.39; 22:10.

(b) Give the same with the budget, and its largest retry ratio. *Checkpoint:* 0, 13, 15.0 %, $27.19; 0.10; 20:13.

(c) The crowd ends at 20:12. Explain why the first run recovers two hours later and the second at once. *Checkpoint:* retries hold the offered load above capacity.

14.10 Application area · 4 min

A team serves a language model on GPU servers like those of case 8: each boots in 8 min, and a fourfold crowd lasts twelve minutes. The manager asks for an autoscaler setting that meets the 2 s objective through such a crowd without refusing any request. Using the chapter's measurements, say what to answer and what the real choice is.

14.11 Application area · 3 min

An operator reads the simulator's knee at 83.2 % and its objective at 87.5 % and proposes to run a production service at an 80 % target. Name two assumptions of the model that would move the real knee to the left, and one that makes the simulated retry storm longer than a service with deadlines would see.

14.12 Review the agent · 5 min

An AI assistant advises on the graded day: "The MPC option includes the server price in its objective, so it minimizes the bill. Keep the suggested settings (Requests metric, admission control, 75 % target) and replace the forecast with MPC, and the day will pass for less than the forecast's $89.49." Find the flaw and say how you would check the claim.