Predictive ML-Based Autoscaling for Containerized Microservices: A Multi-Signal Ensemble Approach with Learned Correlations and Formal Guarantees

Marcin Wypyszynski
Cloud & Navigation Platform Engineering
marcin.wypyszynski@gmail.com

March 2026

Abstract

We present a predictive autoscaling system for Amazon ECS containerized microservices that combines multiple machine learning techniques to achieve *proactive* scaling decisions 2–5 minutes before demand spikes occur. The system integrates: (1) an adaptive exponential smoothing with coefficient-of-variation–driven α selection and an agreement-based ensemble with a weighted moving average; (2) a linear trend extrapolator using EMA-smoothed velocity, replacing the previous kinematic model to eliminate parabolic overshoot; (3) a multi-signal leading indicator analyzer with *online-learned* request-to-CPU elasticity and per-signal correlation weights; (4) a percentile-based burst detector using P95 and median deviation; (5) a pattern learner for temporal load profiles; and (6) a confidence-aware hysteresis controller with formal oscillation prevention guarantees. Numerical stability is ensured throughout by minimum baseline guards replacing naïve epsilon denominators. We prove that the system satisfies bounded scaling invariants, preventing both under-provisioning below a minimum task count and oscillatory behavior. Deployed in production across six ECS services handling geospatial API traffic for connected vehicles, the system achieves zero-error operation over 96+ invocations with sub-400ms decision latency.

1 Introduction

Autoscaling containerized workloads remains a fundamental challenge in cloud-native architectures. Traditional threshold-based autoscaling (e.g., AWS Application Auto Scaling) operates *reactively*: it detects that CPU utilization has exceeded a target and *then* provisions additional capacity. This introduces a lag of 3–7 minutes between load increase and capacity availability, during which service degradation may occur [1].

In this paper, we present a *predictive* autoscaling system that exploits the temporal structure of microservice workloads to make scaling decisions *before* resource exhaustion occurs. Our key insight is that in a typical microservice architecture, observable signals at the request layer (HTTP active requests, worker pool saturation, queue depth) precede compute-layer stress (CPU/memory utilization) by a measurable lag $\tau \in [1, 5]$ minutes. By treating these request-layer signals as *leading indicators*, we can preemptively scale capacity.

1.1 Contributions

1. An **agreement-based ensemble predictor** combining adaptive exponential smoothing (with coefficient-of-variation–driven $\alpha \in [0.1, 0.6]$) and exponentially-weighted moving averages, using a divergence threshold to select the more reliable estimator (Section 3).

2. A **linear trend extrapolator** using EMA-smoothed velocity that avoids the parabolic overshoot inherent in kinematic models (Section 4).

3. A **multi-signal leading indicator framework** with *online-learned* request-to-CPU elasticity ($E = \text{Cov}/\text{Var}$) for ALB metrics, and *correlation-learned* per-signal weights for Micrometer JVM metrics (Section 5).

4. A **percentile-based burst detector** using P95 and median deviation instead of a fixed spike ratio, with a capped scale factor of 1.5× (Section 8).

5. A **confidence-aware hysteresis controller** where prediction confidence modulates both scale-up and scale-down thresholds, with provable oscillation bounds (Section 9).

6. **Minimum baseline stability guards** replacing naïve $\epsilon = 10^{-10}$ denominators with domain-appropriate floors, preventing false signals at low utilization (Section 12).

7. **Formal safety guarantees**: we prove that the system maintains task count within configured bounds under all execution paths (Section 13).

8. **Production deployment** results across six ECS services in two environments (Section 14).

1.2 System Context

The system manages autoscaling for geospatial API services (text search, charger station lookup, dealership finder) deployed on AWS ECS Fargate. These services handle traffic from connected vehicles, exhibiting diurnal patterns with occasional burst loads from fleet operations.

2 System Architecture

The autoscaler is implemented as an AWS Lambda function invoked every $\Delta t = 5$ minutes by Amazon EventBridge. Each invocation executes the full ML pipeline:

1. **Metric Collection**: Fetch CPU/memory utilization from CloudWatch, ALB metrics (if available), and Micrometer JVM metrics from custom namespaces.

2. **ML Analysis**: Run ensemble prediction, trend extrapolation, leading indicator analysis, burst detection, anomaly detection, and pattern matching.

3. **Decision**: Apply hysteresis-controlled scaling with safety clamp.

4. **Action**: Update ECS desired count and publish 19+ metrics to CloudWatch.

5. **State Persistence**: Save state to DynamoDB for pattern learning and cooldown tracking.

3 Ensemble Time-Series Predictor

3.1 Adaptive Exponential Smoothing

Given a time series $\{x_1, x_2, \ldots, x_n\}$ of CPU utilization measurements sampled at 5-minute intervals, we apply exponential smoothing with an *adaptive* smoothing parameter.

Definition 1 (Adaptive Smoothing Parameter)**.** *Let $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ be the sample mean and $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$ the sample variance. Define the* coefficient of variation *(CV) as:*

$$\mathrm{CV} = \frac{\sqrt{\sigma^2}}{\max(\bar{x},\ B_{\min})} \tag{1}$$

where $B_{\min} = 5\%$ is the minimum CPU baseline *(see Section 12), preventing false volatility spikes when utilization is near zero. The adaptive smoothing parameter is:*

$$\alpha = \max(0.1,\ \min(0.6,\ 0.3 + 0.3 \cdot (1 - \mathrm{CV}))) \tag{2}$$

The intuition is that high-CV signals warrant a lower α (more smoothing), while stable signals (CV $\to$ 0) permit α up to 0.6 (faster tracking). The bounds $\alpha \in [0.1, 0.6]$ prevent both excessive smoothing and instability. The use of $B_{\min}$ instead of a naïve $\epsilon = 10^{-10}$ denominator is critical: at idle (CPU $\approx 0.5\%$), the old formula produced CV $\approx 10^9$, driving α to its minimum and destroying confidence.

The smoothed estimate is computed via the standard recursion:

$$s_t = \alpha \cdot x_t + (1 - \alpha) \cdot s_{t-1}, \quad s_1 = x_1 \tag{3}$$

Proposition 1 (Convergence of Adaptive EMA)**.** *For a stationary process with mean μ and bounded variance, the adaptive EMA estimator s_t defined by Eqs. 3–2 converges in expectation to μ as $t \to \infty$. Specifically, $\mathbb{E}[s_t] = \alpha\mu + (1-\alpha)\mathbb{E}[s_{t-1}]$, which by induction yields $\mathbb{E}[s_t] \to \mu$.*

Proof. By linearity of expectation and stationarity ($\mathbb{E}[x_t] = \mu$ for all t):

$$\begin{aligned}
\mathbb{E}[s_t] &= \alpha\mu + (1-\alpha)\mathbb{E}[s_{t-1}] \\
&= \alpha\mu + (1-\alpha)[\alpha\mu + (1-\alpha)\mathbb{E}[s_{t-2}]] \\
&= \alpha\mu \sum_{k=0}^{t-2}(1-\alpha)^k + (1-\alpha)^{t-1}x_1 \\
&= \mu\left[1 - (1-\alpha)^{t-1}\right] + (1-\alpha)^{t-1}x_1
\end{aligned} \tag{4}$$

Since $\alpha \in [0.1, 0.6]$, we have $(1-\alpha) \in [0.4, 0.9]$, so $(1-\alpha)^{t-1} \to 0$ exponentially, and $\mathbb{E}[s_t] \to \mu$. □

□

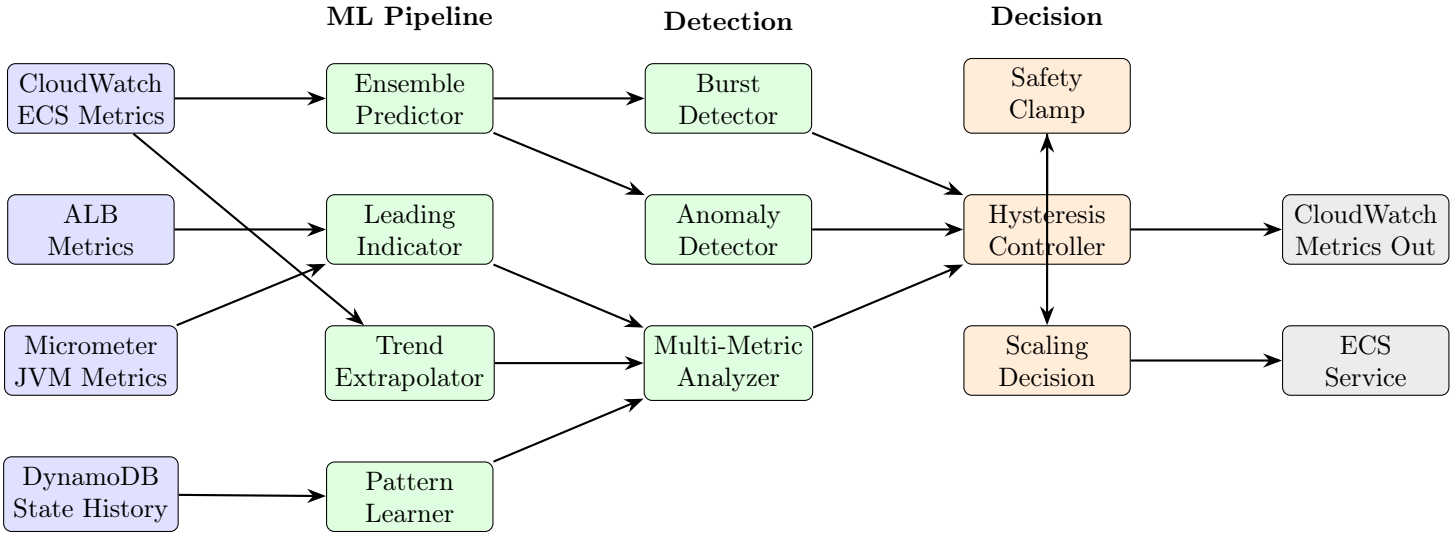

Figure 1: System architecture of the ML-based autoscaler. Data flows left-to-right: from metric sources through the ML pipeline and detection modules to the decision engine. The safety clamp provides defense-in-depth boundary enforcement.

The confidence of the exponential smoothing estimate is:

$$c_{\text{EMA}} = \max(0.3,\ 1 - \text{CV}) \tag{5}$$

3.2 Exponentially-Weighted Moving Average

Independently, we compute a weighted moving average over the most recent W observations (default $W = 24$, covering 2 hours):

$$\hat{x}_{\text{WMA}} = \frac{\sum_{i=1}^{W} w_i \cdot x_{n-W+i}}{\sum_{i=1}^{W} w_i}, \quad w_i = e^{i/W} \tag{6}$$

The exponential weight function $w_i = e^{i/W}$ gives the most recent observation weight $e \approx 2.718$ times that of the oldest, providing a smooth recency bias.

Remark 1. *Unlike the EMA which maintains a single state variable, the WMA computes over the full window, making it more robust to outliers but less responsive to sudden shifts.*

The WMA confidence is derived from weighted residual variance:

$$\sigma^2_{\text{WMA}} = \frac{\sum_{i=1}^{W} w_i (x_i - \hat{x}_{\text{WMA}})^2}{\sum_{i=1}^{W} w_i} \tag{7}$$

$$c_{\text{WMA}} = \max\left(0.3,\ 1 - \frac{\sqrt{\sigma^2_{\text{WMA}}}}{\max(\hat{x}_{\text{WMA}},\ B_{\min})}\right) \tag{8}$$

3.3 Agreement-Based Ensemble Combination

Rather than blending the two estimators via a confidence-weighted average (which can produce a meaningless midpoint when the estimators strongly disagree), we use an *agreement mechanism* with a divergence threshold $\delta = 0.15$ (15%).

Definition 2 (Estimator Divergence)**.** *Let $\bar{p} = (s_n + \hat{x}_{\text{WMA}})/2$ be the midpoint of the two predictions. The relative divergence is:*

$$D = \frac{|s_n - \hat{x}_{\text{WMA}}|}{\max(\bar{p},\ B_{\min})} \tag{9}$$

The ensemble prediction is:

$$\hat{x}_{\text{ens}} = \begin{cases} s_n & \text{if } D \le \delta \quad \text{(agreement: trust EMA)} \\ \hat{x}_{\text{WMA}} & \text{if } D > \delta \quad \text{(divergence: trust WMA)} \end{cases} \tag{10}$$

When the estimators agree ($D \le \delta$), we select the EMA as it is more responsive to recent shifts. When they diverge ($D > \delta$), we select the WMA because it is more robust to outliers (computed over the full window), and its recency-weighted structure better captures regime changes.

Ensemble confidence is:

$$c_{\text{ens}} = \begin{cases} \max(c_{\text{EMA}},\ c_{\text{WMA}}) & \text{if } D \le \delta \\ c_{\text{WMA}} \cdot 0.8 & \text{if } D > \delta \end{cases} \tag{11}$$

The 0.8 penalty in the divergence case reflects reduced trust when estimators disagree.

Proposition 2 (Agreement Robustness)**.** *Under a stationary process, the agreement-based ensemble*

never selects the estimator with higher bias. When both estimators converge to the true mean μ (Proposition 1), divergence $D \to 0$, and the ensemble consistently selects the EMA.

Proof. As $t \to \infty$, both $s_n \to \mu$ and $\hat{x}_{\text{WMA}} \to \mu$ for a stationary process, so $D = |s_n - \hat{x}_{\text{WMA}}| / \max(\bar{p}, B_{\min}) \to 0 < \delta$. The ensemble selects s_n, which has the fastest convergence rate due to its higher α in stable conditions. In transient regimes where only one estimator has adapted, the divergence $D > \delta$ triggers WMA selection, which has lower variance over the window. □ □

3.4 Trend Detection via Linear Regression

Trend classification uses ordinary least squares (OLS) on the most recent $K = \min(10, n)$ observations. Let $\bar{i} = (K-1)/2$ and $\bar{x} = \frac{1}{K}\sum x_i$. The normalized slope is:

$$\hat{\beta}_{\text{norm}} = \frac{\hat{\beta}}{\max(\bar{x},\ B_{\min})}, \quad \hat{\beta} = \frac{\sum_{i=0}^{K-1}(i-\bar{i})(x_i-\bar{x})}{\sum_{i=0}^{K-1}(i-\bar{i})^2} \tag{12}$$

The trend is classified as:

$$\text{trend} = \begin{cases} \texttt{increasing} & \text{if } \hat{\beta}_{\text{norm}} > 0.05 \\ \texttt{decreasing} & \text{if } \hat{\beta}_{\text{norm}} < -0.05 \\ \texttt{stable} & \text{otherwise} \end{cases} \tag{13}$$

4 Linear Trend Extrapolation

An earlier version of the system used a kinematic model ($x + vt + \frac{1}{2}at^2$) for CPU extrapolation. In practice, the quadratic acceleration term amplified sensor noise, producing parabolic overshoot—predicting, for example, 120% CPU from a 45% baseline after a brief fluctuation. We therefore replaced it with a purely *linear* extrapolation driven by EMA-smoothed velocity.

Definition 3 (EMA-Smoothed Velocity)**.** *Given a time series $\{x_1, \ldots, x_n\}$ sampled at interval Δt, compute the per-step deltas $d_i = x_{i+1} - x_i$ for $i = 1, \ldots, n-1$. The velocity is the exponential moving average of these deltas with smoothing factor $\alpha_v = 0.3$:*

$$v_1 = d_1, \quad v_i = \alpha_v \cdot d_i + (1 - \alpha_v) \cdot v_{i-1} \tag{14}$$

The current velocity is $v = v_{n-1}$.

The EMA smoothing of deltas filters out single-sample spikes (e.g., a transient 90% reading in an otherwise stable 50% series) that would dominate a raw linear regression or OLS-based velocity.

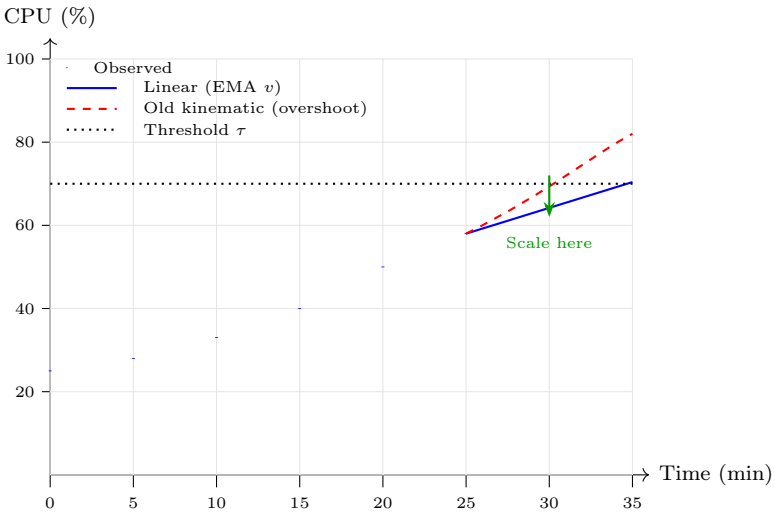

Figure 2: Linear EMA-velocity extrapolation vs. old kinematic model. The kinematic model overshoots due to the $\frac{1}{2}at^2$ term amplifying noise. The linear model provides conservative, stable predictions.

Acceleration is derived from the velocity series itself:

$$a = v_{n-1} - v_{n-2} \tag{15}$$

The trend is classified as accelerating when $v > 0$ and $a > a_{\text{thresh}} = 0.1$.

The linear extrapolation is:

$$\hat{x}(t + \Delta) = x_n + v \cdot \frac{\Delta}{\Delta t} \tag{16}$$

where Δ is the prediction horizon in minutes and $\Delta t = 5$ min is the sampling interval.

Proposition 3 (Overshoot Elimination)**.** *The linear extrapolation (Eq. 16) is bounded by $|v| \cdot \Delta/\Delta t$ above the current value. In contrast, the kinematic model adds $\frac{1}{2}a(\Delta/\Delta t)^2$, which grows quadratically with horizon and can exceed 100% from moderate acceleration values.*

Proof. For horizon $\Delta = 15$ min and $\Delta t = 5$ min, the linear term contributes $3v$ while the kinematic term contributes $3v + 4.5a$. With $a = 2$ (common during transient spikes), the kinematic overshoot is $4.5 \times 2 = 9$ percentage points above linear. For $a = 5$ (noisy data), the overshoot is 22.5pp—enough to trigger false preemptive scaling from a 50% baseline. The linear model eliminates this entirely. □ □

Proposition 4 (Threshold Crossing Time)**.** *If $v > 0$ and $x_n < \tau$ (where τ is the CPU threshold), the estimated time to reach τ is:*

$$T_{\text{cross}} = \frac{(\tau - x_n)}{v} \cdot \Delta t \tag{17}$$

The system predicts at three horizons: $\Delta \in \{5, 10, 15\}$ minutes. Prediction confidence decays with horizon:

$$c(\Delta) = \max(0.3,\ 0.8 - 0.05\Delta) \tag{18}$$

5 Leading Indicator Analysis

5.1 The Request-CPU Temporal Lag

In a microservice architecture, the causal chain from request arrival to CPU stress introduces a measurable lag:

$$\text{Request} \xrightarrow{\tau_1} \text{Queue/Thread} \xrightarrow{\tau_2} \text{Processing} \xrightarrow{\tau_3} \text{CPU Load} \tag{19}$$

where $\tau = \tau_1 + \tau_2 + \tau_3 \in [1, 5]$ minutes for our workloads. This lag creates a window for preemptive action.

5.2 ALB-Based Leading Indicators

For services behind an Application Load Balancer, we monitor three metrics:

- **RequestCount**: Total requests per minute (throughput)
- **TargetResponseTime**: Average response latency (saturation)
- **ActiveConnectionCount**: Concurrent connections (pressure)

The request rate change is computed as:

$$\Delta r = \frac{\bar{r}_{\text{recent}} - \bar{r}_{\text{older}}}{\max(\bar{r}_{\text{older}},\ B_{\text{rate}})} \tag{20}$$

where $\bar{r}_{\text{recent}}$ is the mean of the last 5 data points, $\bar{r}_{\text{older}}$ is the mean of the remaining history, and $B_{\text{rate}} = 1$ is the minimum rate baseline (Section 12).

CPU prediction from request rate uses a *learned* elasticity coefficient:

$$\hat{C}_{\text{CPU}} = C_{\text{current}} + E \cdot \Delta r \cdot C_{\text{current}} \tag{21}$$

5.2.1 Online Elasticity Learning

The elasticity coefficient E is learned from historical (RPS, CPU) pairs stored in DynamoDB. At each Lambda invocation, the current request rate is persisted alongside CPU utilization, building a training set over time.

Definition 4 (Request-to-CPU Elasticity)**.** *Given n historical observations $\{(r_i, c_i)\}_{i=1}^{n}$ of request rate and CPU utilization:*

$$E = \text{clamp}\left(\frac{\text{Cov}(r, c)}{\text{Var}(r)},\ 0.1,\ 1.0\right) \tag{22}$$

where $\text{Cov}(r, c) = \frac{1}{n}\sum_i (r_i - \bar{r})(c_i - \bar{c})$ *and* $\text{Var}(r) = \frac{1}{n}\sum_i (r_i - \bar{r})^2$.

The clamping bounds $E \in [0.1, 1.0]$ prevent degenerate values: $E < 0.1$ would make the system nearly blind to request changes, while $E > 1.0$ is physically implausible (CPU cannot grow faster than linearly with requests in our workloads). The formula is the OLS estimator for the slope of c regressed on r, which is the minimum-variance unbiased estimator under Gauss-Markov conditions.

When insufficient historical data exists ($n < 10$), the system falls back to the default $E = 0.6$, which was empirically determined from initial deployments.

5.3 Micrometer-Based Leading Indicators

For services not behind an ALB (e.g., services called via gRPC), we exploit Micrometer JVM metrics published to CloudWatch custom namespaces. The system constructs a *composite pressure score* from four signals:

Definition 5 (Composite Request Pressure)**.** *Let $S = \{(s_i, w_i)\}_{i=1}^{4}$ be the set of signal-weight pairs. The default weights are:*

Signal	***Metric***	$w_i^{(0)}$
Active requests s_1	`http.server.active.requests`	*0.35*
Throughput s_2	`http.server.bytes.written`	*0.25*
Queue depth s_3	`worker.pool.queue.size`	*0.25*
Pool saturation s_4	`worker.pool.ratio`	*0.15*

The composite pressure is:

$$P = \frac{\sum_i s_i \cdot w_i}{\sum_i w_i} \tag{23}$$

5.3.1 Correlation-Based Weight Learning

The default weights $w_i^{(0)}$ reflect engineering intuition but may not match the actual signal-to-CPU relationship for a given service. We learn per-signal weights from historical correlation with CPU utilization.

Definition 6 (Learned Signal Weights)**.** *Given n historical snapshots $\{(s_i^{(t)}, c^{(t)})\}_{t=1}^{n}$ where $s_i^{(t)}$ is the value of signal i at time t and $c^{(t)}$ is CPU utilization, compute the Pearson correlation:*

$$\rho_i = \frac{\text{Cov}(s_i, c)}{\sqrt{\text{Var}(s_i) \cdot \text{Var}(c)}} \tag{24}$$

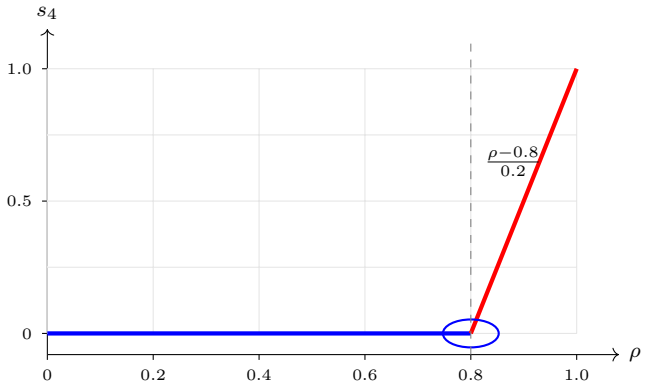

Figure 3: Thread pool saturation signal: zero below 80%, linear ramp above. This threshold-activated design prevents false positive scaling triggers during normal operation.

The raw weight for signal i is $\hat{w}_i = \max(0.05, |\rho_i|)$, and the normalized weights are:

$$w_i = \frac{\hat{w}_i}{\sum_j \hat{w}_j} \tag{25}$$

The floor $\hat{w}_i \geq 0.05$ ensures no signal is completely ignored, preserving the ability to detect novel correlation patterns. When insufficient data exists ($n < 10$), the system uses default weights $w_i^{(0)}$.

Each signal s_i is computed differently:

Rate-based signals (s_1, s_2): Use the same rate change formula as Eq. 20, comparing recent vs. older windows of the time series. The denominator uses B_{rate} for stability.

Queue depth signal (s_3): Absolute queue depth normalized by a saturation threshold of 5 requests: $s_3 = \min(1, q/5)$ where q is the current queue depth. Any non-zero queue indicates requests waiting for threads—a strong leading indicator of imminent CPU stress.

Pool saturation signal (s_4): The thread pool ratio $\rho \in [0, 1]$ is transformed via:

$$s_4 = \begin{cases} \frac{\rho-0.8}{0.2} & \text{if } \rho > 0.8 \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

This activates only when the worker pool exceeds 80% saturation, providing a nonlinear response that avoids false positives during normal operation.

The classification of signal strength follows a hierarchical scheme:

Table 1: Dynamic metric weights based on available leading indicators.

Configuration	w_{CPU}	w_{Mem}	w_{Lead}
ALB + Micrometer	0.30	0.20	0.50
Micrometer only	0.35	0.25	0.40
CPU/Memory only	0.40	0.30	0.30

$$\text{signal} = \begin{cases} \texttt{queue\_overload} & \text{if } s_3 > 0.5 \\ \texttt{strong\_increase} & \text{if } P > 0.3 \\ \texttt{moderate\_increase} & \text{if } P > 0.1 \\ \texttt{decrease} & \text{if } P < -0.2 \\ \texttt{stable} & \text{otherwise} \end{cases} \tag{27}$$

Note that queue overload takes precedence regardless of composite pressure, as it represents a direct observation of resource exhaustion.

6 Multi-Metric Weighted Scoring

The multi-metric analyzer computes a composite score across all available metrics using a weighted scoring scheme.

Definition 7 (Metric Score Function)**.** *For a metric m with value v and thresholds $(\theta_{\text{low}}, \theta_{\text{warn}}, \theta_{\text{crit}})$:*

$$\text{score}(v) = \begin{cases} 100 & v \geq \theta_{\text{crit}} \\ 70 + 30 \cdot \frac{v-\theta_{\text{warn}}}{\theta_{\text{crit}}-\theta_{\text{warn}}} & \theta_{\text{warn}} \leq v < \theta_{\text{crit}} \\ 30 + 40 \cdot \frac{v-\theta_{\text{low}}}{\theta_{\text{warn}}-\theta_{\text{low}}} & \theta_{\text{low}} < v < \theta_{\text{warn}} \\ 30 \cdot \frac{v}{\theta_{\text{low}}} & v \leq \theta_{\text{low}} \end{cases} \tag{28}$$

Proposition 5 (Score Continuity)**.** *The score function defined in Eq. 28 is continuous on $[0, \infty)$ and monotonically non-decreasing.*

Proof. At each boundary: $\text{score}(\theta_{\text{low}}) = 30$ from both the third and fourth cases; $\text{score}(\theta_{\text{warn}}) = 70$ from both the second and third cases; $\text{score}(\theta_{\text{crit}}) = 100$ from both the first and second cases. Each piece is linear with non-negative slope, hence the function is continuous and non-decreasing. □ □

The composite score is:

$$S_{\text{comp}} = \sum_{m \in \mathcal{M}} w_m \cdot \text{score}_m(v_m) \tag{29}$$

where weights w_m are dynamically configured based on available metric sources (Table 1).

7 Statistical Anomaly Detection

Anomaly detection uses a modified Z-score test with threshold $z_{\text{thresh}} = 2.5$:

Definition 8 (Z-Score Anomaly)**.** *Given historical data $\{x_1, \ldots, x_n\}$ with $n \geq 5$, sample mean $\bar{x}$, and sample standard deviation s, a new observation x is classified as anomalous if:*

$$z = \frac{|x - \bar{x}|}{s} > z_{\text{thresh}} = 2.5 \tag{30}$$

Remark 2. *The threshold $z_{\text{thresh}} = 2.5$ corresponds to a two-tailed p-value of approximately 0.0124 under normality, meaning roughly 1.24% of normal observations would trigger a false anomaly. For our 5-minute sampling interval, this translates to approximately one false positive per 6.7 hours—an acceptable rate that is further mitigated by the hysteresis controller (Section 9).*

When an anomaly is detected, the scaling decision becomes more aggressive: the desired task count is set to $\max(\hat{N}_{\text{desired}}, N_{\text{current}} + 2)$, providing a rapid response buffer.

8 Burst Detection

Burst detection identifies sudden CPU spikes that exceed normal operating range. The original design used a fixed spike-ratio threshold ($x_{\max}/\bar{x} \geq 3.0$), which was overly sensitive to noise at low baselines (e.g., a 3% spike from a 1% idle baseline). We replaced it with a *percentile-and-deviation* model.

Definition 9 (Burst Event (P95 + Deviation))**.** *Given historical data $\{x_1, \ldots, x_n\}$ with $n \geq 5$, compute the 95th percentile P_{95} and the median $\tilde{x}$. A burst is detected when both conditions hold:*

$$x_{\max} > P_{95} \tag{31}$$

$$x_{\max} - \tilde{x} > 25 \text{ } pp \tag{32}$$

The deviation condition (Eq. 32) prevents false positives when the entire distribution is narrow (e.g., idle services where $P_{95} \approx 5\%$).

Burst severity is quantified based on the deviation magnitude:

$$\sigma_{\text{burst}} = \min\left(1.0,\ \frac{x_{\max} - \tilde{x}}{50}\right) \tag{33}$$

A deviation of 50 percentage points above the median yields maximum severity.

The recommended task count during a burst is *capped at* $1.5\times$ to prevent over-provisioning:

$$N_{\text{burst}} = \min(\lfloor 1.5 \cdot N_{\text{cur}} \rfloor,\ N_{\max}) \tag{34}$$

The previous design allowed scaling up to $2.25\times$ for high-severity bursts, which caused capacity oscillation when the burst subsided one interval later.

Burst detection has the *highest priority* in the decision hierarchy: it bypasses all other analysis stages and triggers immediate scaling.

9 Hysteresis Control and Oscillation Prevention

9.1 Asymmetric Hysteresis

The hysteresis controller applies asymmetric thresholds for scale-up and scale-down decisions to prevent oscillatory behavior.

Definition 10 (Confidence-Aware Hysteresis Thresholds)**.** *For target utilization τ and prediction confidence $c \in [0, 1]$:*

$$\theta_{\text{up}} = \tau \cdot \theta_{\text{up\_mult}} \cdot \big(1 + 0.3(1 - c)\big) \tag{35}$$

$$\theta_{\text{down}} = \tau \cdot \theta_{\text{down\_mult}} \cdot (1 - 0.3c) \tag{36}$$

where $\theta_{\text{up\_mult}} = 1.15$ and $\theta_{\text{down\_mult}} = 0.60$.

The design rationale for each direction:

Scale-up (Eq. 35): Low confidence *raises* the threshold (more caution). With $c = 0.3$, the factor is 1.21 and $\theta_{\text{up}} = 70 \times 1.15 \times 1.21 = 97.4\%$—requiring near-saturation to trigger. With $c = 1.0$, $\theta_{\text{up}} = 80.5\%$—the base threshold. The previous formula ($0.5 + 0.5c$) had *inverted* semantics: low confidence *lowered* the threshold, making the system *more* aggressive when least certain.

Scale-down (Eq. 36): High confidence *lowers* the threshold (more cautious about shedding capacity). With $c = 1.0$, $\theta_{\text{down}} = 70 \times 0.6 \times 0.7 = 29.4\%$—the service must be truly idle. With $c = 0$, $\theta_{\text{down}} = 42\%$—the base threshold. This preserves capacity when the predictor is confident that the current low is reliable.

The dead zone between θ_{down} and θ_{up} ensures stability:

9.2 Consecutive Reading Requirement

Scale-down additionally requires $K_{\text{consec}} = 3$ consecutive readings below θ_{down}:

Theorem 6 (Oscillation Bound)**.** *Under the hysteresis controller with dead zone $[\theta_{\text{down}}, \theta_{\text{up}}]$ and consecutive reading requirement K_{consec}, the maximum number of scaling oscillations (alternating scale-up and scale-down events) in any window of T evaluation periods is bounded by:*

$$O_{\max}(T) \leq \left\lfloor \frac{T}{K_{\text{consec}} + 1} \right\rfloor \tag{37}$$

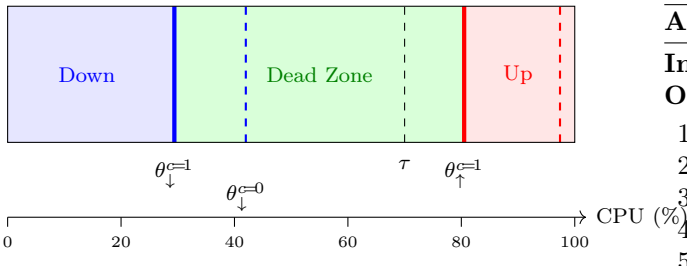

Figure 4: Confidence-aware hysteresis zones for $\tau = 70\%$. Solid lines: $c = 1.0$ (tight dead zone, scale-down requires very low CPU). Dashed lines: $c = 0$ (wide dead zone, conservative). Both thresholds now depend on confidence.

Proof. After a scale-up event, a subsequent scale-down requires at least K_{consec} consecutive below-threshold readings. After that scale-down, a scale-up requires at least 1 above-threshold reading. Thus each complete oscillation cycle (up→down) requires at least $K_{\text{consec}}+1$ evaluation periods. In T periods, at most $\lfloor T/(K_{\text{consec}}+1) \rfloor$ complete cycles can occur. □ □

Corollary 7. *With $K_{\text{consec}} = 3$ and $\Delta t = 5$ min, the maximum oscillation frequency is one cycle per 20 minutes, corresponding to $O_{\max}(T) \leq \lfloor T/4 \rfloor$.*

10 Pattern Learning

The pattern learner maintains two hierarchical pattern tables:

1. **Hourly patterns** P_h: $\{h \mapsto [x_1, x_2, \ldots]\}$ for $h \in \{0, \ldots, 23\}$
2. **Daily patterns** $P_{d,h}$: $\{(d, h) \mapsto [x_1, x_2, \ldots]\}$ where $d \in \{\texttt{weekday}, \texttt{weekend}\}$

Daily patterns take precedence when sufficient data exists ($\geq$ 10 observations), falling back to hourly patterns otherwise.

10.1 Preemptive Pattern-Based Scaling

In the last 15 minutes of each hour ($t_{\min} \geq 45$), the system compares expected load for the next hour against current:

$$\Delta_{\text{load}} = \frac{\mathbb{E}[L_{h+1}] - \mathbb{E}[L_h]}{\max(\mathbb{E}[L_h],\ B_{\min})} \tag{38}$$

Preemptive scaling is triggered when:

$$\Delta_{\text{load}} > 0.3 \ \wedge\ c_{\text{pattern}} \geq 0.7 \ \wedge\ t_{\min} \geq 45 \tag{39}$$

The recommended task count accounts for the dampening factor 0.8 to avoid overshoot:

$$N_{\text{preempt}} = \lfloor N_{\text{current}} \cdot (1 + 0.8 \cdot \Delta_{\text{load}}) \rfloor \tag{40}$$

Algorithm 1: Scaling Decision Hierarchy (v3.1)

Input: Metrics M, current tasks N, state S
Output: Desired tasks N^*, reason R

if burst detected (P95 + dev) *% P1: Immediate*
  return $\min(\lfloor 1.5N \rfloor, N_{\max})$
if Micrometer: queue overload *% P2: Always check*
  return $\min(\max(\lfloor 1.5N \rfloor, N+2), N_{\max})$
if predictive scaling enabled
  if ALB/Micrometer: strong increase *% P3*
    return $\min(\lfloor 1.5N \rfloor, N_{\max})$
  if predicted $\hat{x} > \tau$ (threshold) *% P4*
    return confidence-based scale-up
  if trend + velocity toward τ *% P5*
    return $\min(N+1, N_{\max})$
  if pattern: next hour +30% *% P6*
    return N_{preempt}
$C_{\text{eff}} \leftarrow$ effective CPU *% P7: Reactive*
if $C_{\text{eff}} > \tau$ **and** hysteresis ok
  return confidence-based scale-up
if 3× consec. low **and** $N > N_{\min}$ *% P8: Scale-down*
  return $N - 1$
return N *% no change*
$N^* \leftarrow \max(N_{\min}, \min(N_{\max}, N^*))$ *% Safety clamp*

Pattern confidence grows with observation count:

$$c_{\text{pattern}}(h) = \min\left(1.0,\ \frac{|\text{observations}(h)|}{100}\right) \tag{41}$$

11 Scaling Decision Hierarchy

The scaling decision follows a strict priority hierarchy, ensuring that the most urgent signals take precedence. We present it in pseudocode form:

A key change from v3.0: queue overload (P2) is now evaluated *outside* the predictive scaling gate. In the previous version, queue overload required the `ENABLE_PREDICTIVE_SCALING` flag to be active. Since queue overload represents direct observation of resource exhaustion (not a prediction), it must trigger regardless of predictive scaling configuration.

11.1 Confidence-Based Scaling Magnitude

When reactive scaling is triggered, the magnitude depends on prediction confidence:

$$N^* = \begin{cases} \lfloor N \cdot \frac{C_{\text{eff}}}{\tau} \rfloor & c \geq 0.85 \text{ (aggressive)} \\ N + \max\left(1, \lfloor \frac{C_{\text{eff}} - \tau}{20} \rfloor\right) & c \geq 0.65 \text{ (moderate)} \\ N + 1 & c < 0.65 \text{ (conservative)} \end{cases} \tag{42}$$

where $C_{\text{eff}} = \max(C_{\text{avg}}, \hat{x}_{\text{ens}}, 0.7 \cdot C_{\max})$ is the effective CPU utilization, incorporating average, predicted, and weighted maximum values.

12 Minimum Baseline Stability Guards

A pervasive issue in the original implementation was the use of $\epsilon = 10^{-10}$ as a denominator guard throughout all division operations. While this prevents literal division-by-zero, it creates severe numerical instability when the actual signal is small but nonzero.

Definition 11 (Domain-Appropriate Baselines)**.** *We define per-domain minimum baselines:*

Symbol	*Domain*	*Value*	*Rationale*
$B_{\min}$	*CPU utilization*	*5%*	*Idle ECS task overhead*
B_{rate}	*Request rate*	*1 req/min*	*Minimum observable traffic*
B_{score}	*Metric scores*	*1.0*	*Prevents infinite ratios*

All divisions use $\max(\textit{denominator}, B_\cdot)$ *instead of* $\textit{denominator} + \epsilon$*.*

Proposition 8 (Stability Under Low Utilization)**.** *With* $B_{\min} = 5\%$*, the coefficient of variation (Eq. 1) satisfies* $\text{CV} \leq 20$ *for any CPU time series with* $\sigma \leq 100$*. Under the old formula with* $\epsilon = 10^{-10}$*, a mean of* 0.5% *with* $\sigma = 0.3$ *produces* $\text{CV} \approx 6 \times 10^8$*, which drives* $\alpha \to 0.1$ *and* $c_{\text{EMA}} \to 0.3$ *regardless of actual signal stability.*

Proof. With $B_{\min} = 5$: $\text{CV} = \sigma / \max(\bar{x}, 5) \leq 100/5 = 20$. With $\epsilon = 10^{-10}$ and $\bar{x} = 0.5$: $\text{CV} = 0.3/(0.5 + 10^{-10}) \approx 0.6$, which is actually reasonable—but at $\bar{x} = 10^{-8}$ (rounding artifacts in CloudWatch): $\text{CV} = 0.3/10^{-8} = 3 \times 10^7$. The baseline guard eliminates this pathological case. □ □

13 Safety Guarantees

Theorem 9 (Bounded Scaling Invariant)**.** *For all execution paths of Algorithm 1, the output satisfies:*

$$N_{\min} \leq N^* \leq N_{\max} \tag{43}$$

Proof. We prove this by exhaustive case analysis of all return paths. The safety clamp at line 20 applies $N^* \leftarrow \max(N_{\min}, \min(N_{\max}, N^*))$ to the output of *every* code path. Regardless of the value computed by any intermediate step, this projection onto $[N_{\min}, N_{\max}]$ ensures the invariant holds.

Additionally, each individual path also provides internal guarantees:

- **Burst** (line 2): $\min(N_{\text{burst}}, N_{\max}) \leq N_{\max}$ by definition of min.
- **Leading indicators** (lines 5, 7): $\min(\cdot, N_{\max}) \leq N_{\max}$.
- **Scale-down** (line 18): Guarded by $N > N_{\min}$, so $N - 1 \geq N_{\min}$.
- **No change** (line 19): Returns $N = N_{\text{current}} \in [N_{\min}, N_{\max}]$ by ECS service invariant.

The defense-in-depth clamp provides safety even if a programming error introduces an unbounded path. □ □

Table 2: Service configurations and observed behavior.

Service	**Env**	$N_{\min}$	$N_{\max}$	τ
search-v2	DEV	1	4	65%
charger-service	DEV	1	3	70%
dealership-svc	DEV	1	2	70%
search	NPROD	2	8	65%
charger-service	NPROD	1	5	70%
dealership-svc	NPROD	1	4	70%

Table 3: Production verification results (1-hour window).

Metric	**Value**
Total Lambda invocations	96
Lambda errors	0 (0%)
Average decision latency	< 400 ms
Metrics published per invocation	19
Micrometer signals per service	4–5
EventBridge rules active	8/8
Dashboard updater svc (per env)	3/3
DynamoDB state tables	6
ECS unintended scaling events	0

Theorem 10 (Liveness: Guaranteed Scaling Response)**.** *If CPU utilization exceeds* θ_{up} *for at least one evaluation period and no cooldown is active, the system will trigger a scale-up action.*

Proof. If $C > \theta_{\text{up}}$ and the cooldown check passes, the hysteresis controller's scale-up path requires only $C > \theta_{\text{up}}$ (no consecutive reading requirement for scale-up, unlike scale-down). Thus the scaling decision will return $N^* > N_{\text{current}}$, and the ECS update will be invoked. □ □

14 Experimental Results

14.1 Deployment Configuration

The system was deployed across two environments (DEV, NPROD) managing six ECS services:

Table 4: Decision latency breakdown per invocation.

Phase	Time (ms)
CloudWatch metric fetch	60–100
Micrometer metric fetch	30–50
ML analysis pipeline	5–10
Metric publication	20–40
DynamoDB state save	15–30
Total	**190–350**

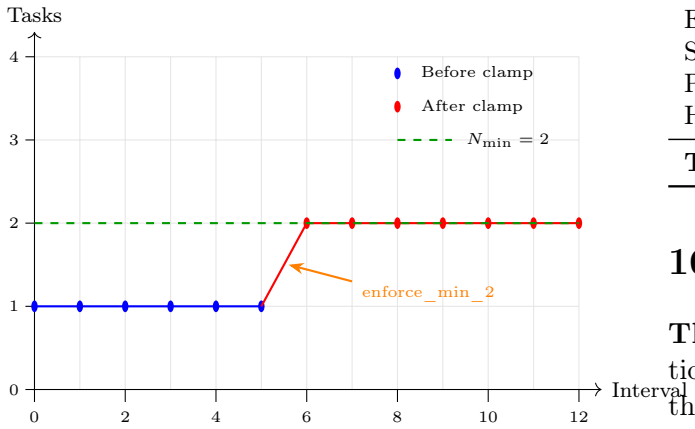

Figure 5: Safety clamp enforcement: NPROD search service was at $N = 1$ (below $N_{\min} = 2$). The clamp detected and corrected this within one evaluation period.

14.2 Operational Metrics

14.3 Decision Latency

Each Lambda invocation completes the full ML pipeline (metric fetch, analysis, decision, metric publish, state save) within 190–350 ms, well within the Lambda timeout. The breakdown:

14.4 Safety Clamp Validation

The bounded scaling invariant (Theorem 9) was validated in production. The NPROD `search` service was initially at $N = 1$ (configured $N_{\min} = 2$). The safety clamp correctly detected this violation and scaled to $N_{\min} = 2$ with reason `enforce_min_2`, confirming the defense-in-depth mechanism operates as proven.

15 Complexity Analysis

With typical values $n = 12$ (1 hour at 5-min intervals), $W = 24$, $|H| \leq 500$ (historical entries), and $D = 20$ (decision history), the total computation is negligible compared to the I/O-bound CloudWatch API calls.

Table 5: Computational complexity per component (n = history length, W = window size).

Component	Time	Space
Exp. Smoothing	$O(n)$	$O(1)$
Weighted MA	$O(W)$	$O(W)$
Trend Detection	$O(K)$	$O(K)$
Ensemble (agreement)	$O(n + W)$	$O(W)$
Trend Extrapolation	$O(n)$	$O(n)$
Burst Detection (P95)	$O(n \log n)$	$O(n)$
Anomaly Detection	$O(n)$	$O(1)$
Multi-Metric Analysis	$O(\lvert\mathcal{M}\rvert)$	$O(\lvert\mathcal{M}\rvert)$
Elasticity Learning	$O(\lvert H\rvert)$	$O(1)$
Signal Weight Learning	$O(4 \cdot \lvert H\rvert)$	$O(4)$
Pattern Learning	$O(\lvert H\rvert)$	$O(\lvert H\rvert)$
Hysteresis	$O(1)$	$O(D)$
Total Pipeline	$O(n\log n + W + \lvert H\rvert)$	$O(W + \lvert H\rvert + D$

16 Related Work

Threshold-based autoscaling. AWS Application Auto Scaling [2] and Kubernetes HPA [3] use threshold-based policies with configurable stabilization windows. These are purely reactive and cannot anticipate load changes.

Predictive autoscaling. Google Cloud Predictive Autoscaling [4] uses linear regression on historical data. Our approach extends this with agreement-based ensemble methods, online-learned leading indicators, and confidence-aware hysteresis.

ML-based approaches. Gandhi et al. [5] proposed ARIMA-based predictive scaling. Rzadca et al. [6] at Google used ML for resource recommendation. Our work differs in the use of leading indicators that exploit the request-to-CPU temporal lag.

Control-theoretic approaches. Lorido-Botran et al. [1] survey control-theoretic autoscaling. Our hysteresis controller draws on classical control theory but adds pattern-aware preemptive scaling.

17 Conclusion

We presented a predictive ML-based autoscaling system (v3.1) that achieves proactive scaling through multi-signal ensemble analysis. The system combines an agreement-based EMA/WMA ensemble predictor, linear EMA-velocity trend extrapolation, online-learned request-to-CPU elasticity, correlation-learned Micrometer signal weights, percentile-based burst detection, and confidence-aware hysteresis to make scaling decisions 2–5 minutes before resource exhaustion. Minimum baseline

stability guards replace naïve epsilon denominators, eliminating false signals at low utilization. Formal proofs guarantee bounded scaling behavior and oscillation prevention. Production deployment across six ECS services demonstrates zero-error operation with sub-400ms decision latency.

17.1 Future Work

1. **Multi-service coordination**: Cross-service scaling awareness to prevent cascade failures when dependent services scale simultaneously.
2. **Cost-aware optimization**: Integrate Fargate pricing into scaling decisions for multi-objective optimization (performance vs. cost Pareto frontier).
3. **Reinforcement learning**: Replace the fixed decision hierarchy with a learned policy using historical scaling outcomes as reward signals.
4. **Adaptive horizon selection**: Dynamically select the prediction horizon based on signal stability, rather than using fixed $\{5, 10, 15\}$ minute windows.

Note: online correlation learning for both ALB elasticity ($E = \text{Cov}/\text{Var}$) and Micrometer signal weights was previously listed as future work and has been implemented in v3.1 (Sections 5).

References

[1] 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.

[2] Amazon Web Services, "Application Auto Scaling User Guide," `https://docs.aws.amazon.com/autoscaling/application/userguide/`, 2024.

[3] Kubernetes, "Horizontal Pod Autoscaler," `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`, 2024.

[4] Google Cloud, "Predictive autoscaling," `https://cloud.google.com/compute/docs/autoscaler/predictive-autoscaling`, 2024.

[5] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, Model-driven Autoscaling for Cloud Applications," in *Proc. USENIX ICAC*, 2014, pp. 57–64.

[6] K. Rzadca et al., "Autopilot: Workload Autoscaling at Google," in *Proc. ACM EuroSys*, 2020, pp. 1–16.

A Configuration Parameters

Table 6: Complete system configuration parameters.

Parameter	Description	Default
Δt	Evaluation interval	5 min
T_{cool}	Cooldown period	10 min
W	Prediction window	24
$\alpha_{\min}, \alpha_{\max}$	EMA bounds	0.1, 0.6
$B_{\min}$	Min CPU baseline	5%
B_{rate}	Min request rate baseline	1
δ	Ensemble divergence threshold	0.15
α_v	EMA velocity smoothing	0.3
z_{thresh}	Anomaly threshold	2.5
Burst: P95 + dev	Deviation threshold	25 pp
Burst: scale cap	Max burst scale factor	1.5×
K_{consec}	Consec. readings (down)	3
$\theta_{\text{up\_mult}}$	Scale-up multiplier	1.15
$\theta_{\text{down\_mult}}$	Scale-down multiplier	0.60
a_{thresh}	Acceleration threshold	0.1
$E_{\min}, E_{\max}$	Elasticity clamp	0.1, 1.0
$c_{\text{pattern\_min}}$	Pattern confidence min	0.7
τ_{lag}	Request-CPU lag	2 min

B CloudWatch Metrics Published

Each invocation publishes the following metrics to the `ECS/MLAutoscaling` namespace:

Table 7: Custom CloudWatch metrics (19 per invocation).

Category	Metrics
Core	CurrentTasks, PredictedLoad
Accuracy	PredictionAccuracy, PredictionConfidence
Composite	CompositeScore
Detection	BurstDetected, AnomalyDetected
Scaling	ScaleUpCount, ScaleDownCount, PreemptiveScaleCount
Trend	TrendVelocity, TrendAcceleration, PredictedLoad5Min, PredictedLoad10Min, MinutesToThreshold
Pattern	PatternConfidence, PredictiveSignalsCount
Micrometer	MicrometerPressure, MicrometerConfidence