LSTM-Based Approach to Predictive Cloud Autoscaling

David Yi, Danny Yu, Darren Choe, Justin Li, Cory Pham, Nathan Chiu, Ziyi Chen, Taniqsha Bonde, Chuyuan Wang

I. Introduction

Modern cloud-native services experience unpredictable workload fluctuations, making resource management difficult. Conventional autoscaling is reactive, adding resources only after metrics exceed thresholds, which often results in delayed scaling and inefficiency. This motivates predictive autoscaling, where future demand is estimated from historical patterns. This project develops a data-driven predictive autoscaling framework using an LSTM network to model CPU and memory utilization from Google Borg traces. Our goal is to forecast short-term resource usage and show how proactive scaling can improve system stability while reducing costs.

II. Background

Reactive autoscaling, the default mechanism in modern cloud platforms, adjusts resources only after utilization crosses predefined thresholds. Because compute instances require non-trivial startup time, this delayed response often leads to temporary underprovisioning and oscillatory behavior, prompting operators to overprovision resources as a safety buffer. Predictive autoscaling aims to address these limitations by forecasting future load so that resources can be provisioned ahead of demand. Prior work has explored statistical models like ARIMA and exponential smoothing, but their ability to capture complex traffic patterns is limited. LSTM networks offer a stronger alternative due to their gating mechanisms and memory cells, which enable them to model long–range temporal dependencies and periodic usage patterns common in compute clusters. These characteristics make LSTMs well-suited for short-horizon CPU and memory forecasting, which is critical for improving autoscaling responsiveness and stability.

III. Methodology

A. Data Preprocessing

Our study uses a 400,000-row subset of the Google Borg cluster-usage traces, representing CPU and memory utilization across eight clusters in 2011 and 2019. The majority of work was done with the Borg cluster dataset, however, we decided to switch to the Azure Public dataset later on to avoid the shortcoming of limited data (https://github.com/Azure/AzurePublicDataset). Timestamps in microseconds are converted into 5-minute intervals and sorted chronologically, including time of day, day of week, and a weekend indicator. Each training example is created using window of six timesteps (30 minutes of historical data). Rows with unavailable/invalid values are removed, and outliers are mitigated using interquartile-range (IQR) capping. Numerical features (CPU and memory) are normalized using a RobustScaler and one hot encoded to reduce the effect of skew. The final dataset is split 80/20 into training and test sets.

B. Modeling

We evaluate two autoscaling strategies: a reactive autoscaler, which serves as the industry-standard baseline, and a predictive autoscaler driven by our LSTM-based forecasting model.

a) *Final Model:* To address the limitations of threshold-based systems, we design a predictive autoscaling strategy powered by our LSTM model, TrafficPredictorV2, which forecasts CPU and memory utilization 5 minutes into the future.

The predictive policy uses a 2-layer stacked LSTM with 128 hidden units per layer, which processes a sequence of 12 historical timesteps (60 minutes of data). A dropout rate of 0.2 is applied between layers to prevent overfitting, and batch normalization stabilizes the LSTM output. To better leverage the strong autocorrelation present in resource traces, we concatenate the most recent

CPU and memory values from the input sequence with the LSTM's final hidden state. This residual-style connection helps the model learn short-term changes (deltas) rather than absolute values, improving performance on fluctuating workloads. A fully connected prediction head ($130 \rightarrow 64 \rightarrow 32 \rightarrow 2$) with ReLU activations outputs the next-interval CPU and memory utilization values.

b) *Training Process:* The model is trained using the Huber loss function rather than mean squared error, which behaves like MSE for small residuals but transitions to a linear form for larger deviations, making it well-suited for handling the occasional traffic spikes present in the data. Optimization is performed using the AdamW optimizer with a learning rate of 0.002 and weight decay of 1e-5, providing both stable convergence and effective regularization. To further support the optimization process, a cosine annealing learning-rate schedule with warm restarts is used, helping the model escape local minima over the course of training. Training is conducted with a batch size of 16 for up to 100 epochs, with early stopping applied using a patience of 25 epochs to prevent overfitting when validation performance plateaus. The resulting trained model is then integrated into the simulation environment to power the predictive autoscaling policy.

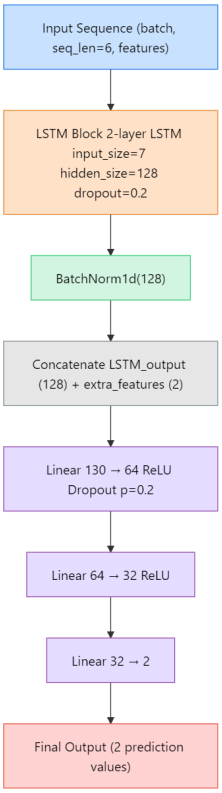

Fig. 1. Model Architecture

C. Simulation

The main objective for the simulation of our autoscaler was to create a rigorous testing environment where we can empirically evaluate the efficacy of our model. We did this by simulating time-steps comparing our model to a baseline reactive autoscaler.

Initially, we developed the simulation engine in Python, which would take a separate set of data from the Google Cluster Trace dataset and step through time minute-by-minute, simulating the state of the cluster according to the inputs. To ensure that this would reflect reality, we implemented an artificial "warm up" latency that considers the booting up of the servers in real life. When a policy issues a scaling action, the resources are delayed by 15 minutes before becoming active. This ensures real-world scenarios since resources cannot be allocated instantly.

We first created a baseline reactive autoscaling policy that incorporated metric tracking to serve as the control group. We also implemented metric tracking classes to identify key indicators such as total cost, scaling events, and under/overprovisioning events. This agent operates under only two

constraints: blindness (knowing only current load) and stability (avoid changing allocation). We use geometric scaling to increase capacity by a fixed percentage once target utilization is exceeded. We do this to treat sudden spikes as potential noise. To prevent oscillation, this policy enforces a strict cooldown window of 3 time steps. In conjunction with this cooldown, the geometric growth of server provisioning results in lackluster provisioning during rapid spikes.

To convert our ML prediction into a real decision, we developed a more sophisticated predictive control policy. We use a direct calculation approach to calculate exactly how many nodes are needed to handle the maximum predicted load in the future horizon, then jumps instantly to this target. Because the LSTM inherently filters noise, we can assume that there is less risk of oscillation, and thus we allow the predictive policy to bypass the cooldown and scale up or down at will.

The predictive policy logic relies on proportional target tracking and hybrid policies. Proportional target tracking calculates the exact amount of servers required to maintain a specific target utilization at a predicted time, allowing the system to jump to the correct size in response to a predicted large spike. For the hybrid policies, we calculate the resource demand based on both the future prediction (t+1) and our current load (t), and take the greater of the two. Even if the model wrongly predicts a traffic drop, the system will never scale below the current required load. However, a standard one-step-ahead prediction proved insufficient due to the boot latency length. To compensate, we implemented autoregression. The LSTM predicts t+1, feeds that prediction as input to predict t+2, and so on until a horizon of h=6. The autoscaling decision is derived from the maximum load within this horizon.

IV. Results

A. Model Training Convergence

The training and test loss curves are shown in Fig. 2. Both curves decrease smoothly over the 80-epoch training run and converge to similar final values between 0.005 and 0.006. The close alignment between training and validation loss indicates that the model does not exhibit overfitting and is able to generalize to unseen workload segments. This confirms the stability of the training procedure and the suitability of the Huber loss for handling occasional load spikes.

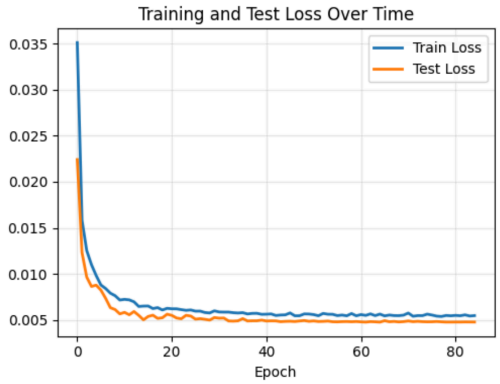

Fig. 2. Loss Graph during Training

B. Prediction Accuracy

In addition to loss metrics, we evaluate the model on the test set using R^2 as another performance metric. As shown in Fig. 3, the predictions follow the overall patterns of the true CPU and memory utilization closely. The model achieves an R^2 of 0.83 for CPU and 0.82 for memory, demonstrating a strong linear relationship between predicted and actual values across typical utilization ranges.

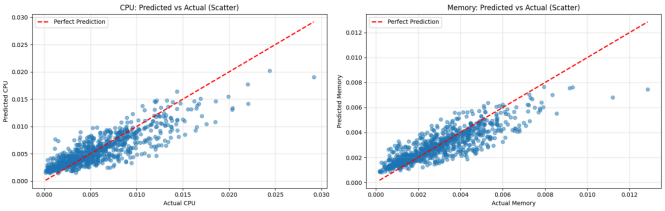

Fig. 3. Predicted vs Actual CPU and Memory Values

The model accurately captures daily cycles, moderate fluctuations, and local trends. This level of predictive performance is sufficient for autoscaling, as the controller primarily requires correct directional movement and approximate magnitude rather than perfect point forecasts.

C. Spike and Failure-Mode Analysis

Although the model performs well under normal operating conditions, its limitations become apparent during rare, high-magnitude utilization surges. As illustrated in Fig. 4, the LSTM consistently underestimates the peaks of large load spikes. Because these spikes occur infrequently and deviate significantly from the typical distribution, the model tends

to regress toward the mean, resulting in conservative predictions.

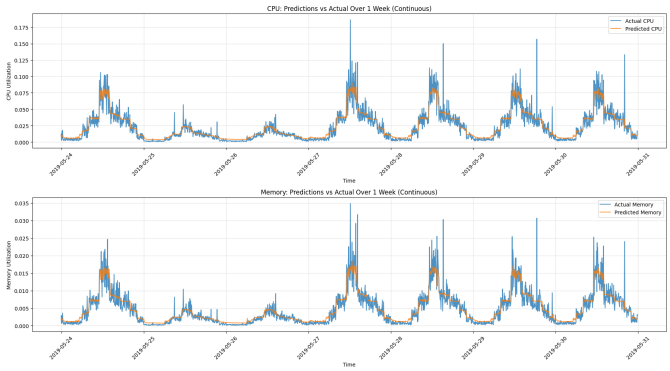

Fig. 4. Performance of model on 5 days of data

This failure mode is expected for sequence models trained on datasets with strong class imbalance between normal and extreme load events. While spike underestimation does not severely degrade average prediction quality, it may affect proactive autoscaling, as the controller may not request additional capacity early enough when sudden bursts occur.

D. Simulation Results

The baseline reactive autoscaling performs adequately, but noticeably lags during spike events. There is also significant overprovisioning, represented by the white gaps between the server capacity and the demand:

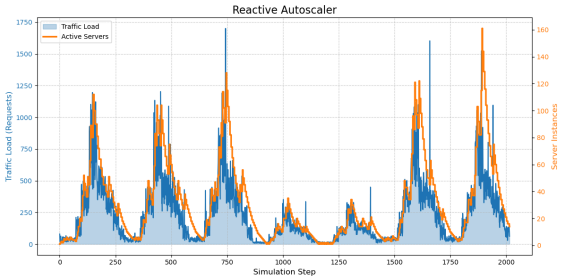

Fig. 5. Reactive Autoscaler Performance

In comparison, the predictive autoscaling follows the traffic far more closely, with significantly reduced overprovisioning as well as more responsive handling of spikes:

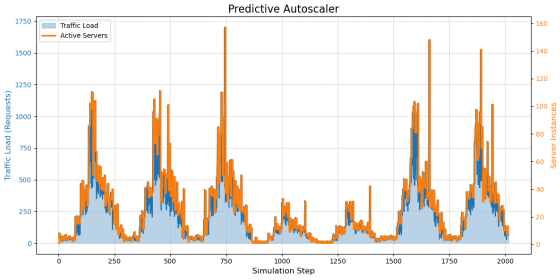

Fig. 6. Predictive Autoscaler Performance

Experimental setup: 2016 data points, each spaced 5 minutes apart, in total representing a full week of traffic. The boot time is set to 3 time steps, or 15 minutes, representing the startup of a heavy enterprise application. We configure the prediction horizon of our model to be 6 steps, or 30 minutes ahead, to make sure the model looks far enough to beat the boot time. We set our safety buffer to 1.45, to slightly overprovision to prioritize stability and reliability.

Over a full simulated week, we can compare the difference in several key metrics:

TABLE I
COMPARISON OF REACTIVE VS. PREDICTIVE AUTOSCALING PERFORMANCE

Metric	Reactive (Baseline)	Predictive (LSTM)	Improvement
Total Cost (Simulated)	$62,919	$62,863	−0.1%
Avg. Resource Utilization	32.1%	39.7%	+23.6%
Crash Events (Count)	30	14	**−53.3%**
Total Unserved Load	1,748	1,626	−7.0%
P99 Latency	5,000.0 ms	25.7 ms	**−99.5%**
Scaling Actions	2,301	6,225	+170.5%
Avg. Instance Count	31.2	31.2	−0.1%

V. DISCUSSION

The results of this study highlight both the strengths and limitations of applying an LSTM-based forecasting model to predictive autoscaling. During data preprocessing, we observed a substantial number of missing, duplicated, or zero-utilization rows in the cluster traces. These artifacts arise because cluster machines often remain in steady-state conditions while idling, executing background processes, or caching memory, resulting in long periods of minimal variation in utilization. Although individual samples are recorded at sub-second gran-

ularity, resource usage tends to change gradually, producing extended sequences of nearly constant values. Swapping to the Azure dataset removed sparsity issues and ensured each cluster retained enough unique temporal structure for meaningful training.

Model evaluation shows that the LSTM captures the majority of temporal patterns, achieving strong predictive performance under typical operating conditions. However, the model consistently underestimates rare, high-magnitude spikes. This behavior is attributable to two factors: (1) the strong class imbalance between normal fluctuations and extreme surges, and (2) the autoregressive tendency of LSTMs to regress toward recent history when provided insufficient examples of rare events. While this limitation does not significantly impact the model's average accuracy, it has important implications for autoscaling. Underestimation of spikes may lead to delayed scale-up actions unless corrective mechanisms are applied.

Under latency constraints, our simulation environment better approximates real-life behavior, making our conclusions more credible for deployment considerations. Because the simulation engine, reactive baseline, and predictive policy all share the same metric-tracking and decision infrastructure, comparisons are fair and consistent. This provides a solid foundation for evaluating future policies under identical conditions. Though we extend predictions over a multi-step horizon (t+6), the future is inherently uncertain: longterm trends, rare spikes, or external events that drive demand spikes are impossible to forecast from historical usage alone. Thus, predictive autoscaling remains inherently limited in guaranteeing performance in all scenarios.

Our predictive autoscaler was able to demonstrate its adaptability by allocating instances to future load values by predicting utilization based on previous data in earlier time periods. Its adequate functionality comes from the fact that our LSTM model was able to predict general patterns and relatively inactive points where usage might not be high. However, it struggles when dealing with sudden high usage values due to the spiky nature of the data. These spikes are varying in size so the predictive autoscaler is not able to completely adapt to every spike, meaning it would have to allocate more instances to generally high utilization spots to act as a safety net, which would help allow resources to be fed through in busy hours. However, this is generally inefficient since a lot of the time, after the spike hits, the predictive autoscaler would just generally keep its instance usage high for the majority of the time to "play it safe", over-committing resources essentially. It was able to do well on relatively inactive periods of the day though.

VI. Conclusion

This study demonstrates that short-horizon workload forecasting using an LSTM model can significantly enhance cloud autoscaling strategies. By leveraging temporal structure in CPU and memory utilization traces, the predictive autoscaler anticipates future deman and initiates scaling actions ahead of time, overcoming inherent limitations of reactive threshold-based policies. The proposed stacked LSTM architecture achieves strong predictive accuracy, and our simulation framework shows that predictive scaling can reduce under-provisioning events and improve system stability.

Although the model underestimates rare spikes, the hybrid scaling logic mitigates the risks associated with conservative predictions. Future work will explore richer temporal models, techniques for capturing extreme load behavior, and integration of probabilistic forecasts to enable uncertainty-aware autoscaling. Overall, this work provides evidence that ML-driven prediction is a viable and effective approach for improving the responsiveness and efficiency of large-scale compute infrastructure.

The most critical finding is that the predictive policy achieves these results without increasing operational costs. This proves that the model improves reliability by spending the same money more intelligently, rather than spending more money to improve performance. In terms of reliability, the predictive policy reduced outages by 53%, while the P99 latency statistic shows that 99% of the user base can experience standard latency given the predictive model. This confirms that for workloads with high boot up lag, reactive logic is mathematically incapable of preventing crashes and latency during spike events. Moreover, we observe that the Predictive policy achieves a higher average

resource utilization (39.7% vs 32.1%), indicating the reactive policy spends significant time overprovisioned, contributing to resource wastage. While the predictive model demonstrates superior uptime and latency metrics, it does introduce significant downsides, namely the increase in scaling actions (170.5%). In a production environment this can lead to increased wear and tear, as well as API rate limiting. Furthermore, it must be noted that the predictive policy does not eliminate failure entirely. While it reacts faster than the reactive baseline, it's predictions are still inherently conservative. Thus, the system exchanges stability for low latency. Still, the predictive approach justifies its high rate of fluctuation by its 200x (or more) improvement in user experience for the same infrastructure cost.

VII. Contribution

David Yi: Developed integration layer connecting LSTM to simulator and created policy logic. Consolidated policies, metric tracking, and simulator into a unified framework.
Danny Yu: Implemented reactive autoscaler and core metric tracking logic for both the reactive and predictive autoscaler.
Darren Choe: Wrote data preprocessing script, produced .csv files for modeling team, added to the discussion/preprocessing section.
Justin Li: Helped with running initial data csv without 10% data filter and assisted with script. Structured presentation slides in succinct deliverable
Cory Pham: Implemented core logic through time, simulating instance usage and scaling based on cpu and memory utilization
Nathan Chiu: Model architecture development and training; created Github repository
Ziyi Chen: Worked on model script and analysis of the training result.
Taniqsha Bonde: Made initial LSTM model for training and testing, did slides for model architecture
Chuyuan Wang: Helped build the data preprocessing script, drafted the report's introduction and data preprocessing sections