National College of Ireland

Forecasting Knative Workload Traffic with Machine Learning to Minimize Cold Starts

MSc Research Project
Cloud Computing

Abigail Mariam Anil
Student ID: x23433698

School of Computing
National College of Ireland

Supervisor: Prof. Aqeel Kazmi

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Abigail Mariam Anil
Student ID:	x23433698
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Prof. Aqeel Kazmi
Submission Due Date:	28/01/2026
Project Title:	Forecasting Knative Workload Traffic with Machine Learning to Minimize Cold Starts
Word Count:	7172
Page Count:	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	28th January 2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Forecasting Knative Workload Traffic with Machine Learning to Minimize Cold Starts

Abigail Mariam Anil
x23433698

Abstract

On serverless platforms like Knative, scaling is automatically configured, however, the default reactive autoscaler reacts only after the load increases, leading to cold starts and latency spikes with bursty and irregular traffic that may not allow most cloud applications to operate efficiently. Previous research has investigated predictive autoscaling systems and hybrid forecasting of cloud systems, yet the end-to-end analysis of machine-learning-based forecasting as part of the Knative autoscaling loop using actual serverless traces has not been performed. This work builds a predictive autoscaling mechanism in Knative based on three predictive models, namely Prophet, LSTM and, Hybrid (Prophet-residuals-plus-LSTM) models, which were trained using the Azure Functions invocation data. Pre-scaling of pods is done based on the predictions before the demand arises. The experimentation results revealed that all the predictive strategies decrease the cold start latency in burst and transition phases. The experiments also show that the forecasting error for LSTM and Hybrid models is comparatively lower than that of Prophet for irregular traces. Prophet demonstrates strong performance for smooth and seasonal workloads, theoretically positioning ML forecasting as a safe extension, instead of being a replacement for reactive auto scaling in Knative. In practice, they maintain low tail latency for latency sensitive functions with relatively low additional resource cost. Overall, the evaluation shows that machine learning based predictive autoscaler reduces cold start latency in burst-heavy workloads without compromising its behaviour and is comparable to the Knative Pod Autoscaler with steady traffic, making it a complementary solution to reactive scaling.

1 Introduction

1.1 Background

Serverless computing, especially Function-as-a-Service (FaaS) platforms, such as Knative, allow automatic scaling and also manage the infrastructure. However, this automation causes a cold start latency, i.e., delays caused during container initialization and loading of functions in the absence of a warm instance.

Cold starts have a big influence on the end-to-end latency of serverless workflows Lee et al. (2021). The majority of the platforms, such as Knative, are reactive-based autoscaling, which only adjusts replicas after load variation has taken place. This strategy does not respond to bursty workloads well, meaning that it leads to under-provisioning when traffic peaks on the system, and extra costs when the traffic starts to decline slowly Hong et al. (2024).

1.2 Motivation and Importance

Applications such as real-time analytics, interactive web services and event-driven pipelines that are latency sensitive are also sensitive to cold starts and rapid surges in workload. Over-provisioning of resources simply is not a feasible option due to the increase in CPU and memory consumption, as well as compromising cost savings, which is a key feature that renders serverless attractive. In this way, responsiveness versus efficiency has become one of the major concerns of serverless autoscaling.

According to recent surveys, there is increased interest in using ML-based cold-start mitigation, but predictive autoscaling in serverless systems has been an under-researched subject Golec et al. (2024). In particular, there is little literature that combines forecasting and Knative autoscaling Tràn and Kim (2023). This work fills this gap, applying the ML-based traffic forecasting to the Knative autoscaling loop while relying on the Azure Functions invocation traces. The purpose here is to minimize the Cold-start time, enhance the response to scaling, and provide a more resource-conscious alternative to the reactive control.

1.3 Research Question and Objectives

This work focuses on answering the Research Question: **How does the machine learning-based forecasting approach influence the proactive scaling of Knative services to reduce the cold start latency compared to the default Knative Pod Autoscaler?**

To answer this research question, the following objectives were formulated:

1. To examine the serverless workload behaviour and trends in the Azure Functions data by identifying temporal patterns like burstiness, idle gaps, seasonality, and predictability to understand scaling challenges in Knative.

2. To create and train forecasting models - namely, Prophet, LSTM, and a hybrid model of Prophet residuals and the LSTM model to predict short-term invocation demand, capturing both seasonal structure and sequential effects on invocation traffic.

3. To deploy the trained forecasting models into the Knative autoscaling system, in a way that it can scale instances proactively based on the predicted future demand.

4. To compare and contrast the predictive autoscaler and the default KPA reactive autoscaler, based on cold start latency, CPU or memory usage, and total resource efficiency.

1.4 Limitations and Structure of Report

This research has a number of limitations. It uses a publicly available Azure Functions invocation dataset, which may not reflect all of the potential heterogeneity of production serverless workloads, e.g. geo-distributed bursts. The forecasting models are used on the assumption that the predictions made are enough to make proactive scaling decisions. Second, due to student account constraints, the evaluation was focused on three representative functions which limits statistical generalizability across workload types. Additionally, the work also does not delve into vertical scaling and resource sizing strategies,

instead it considers horizontal scaling by modifying the number of pods. Cost analysis was not explicitly conducted; although resource usage was measured, cost-performance trade-offs are not quantified. Lastly, the conservative pre-scaling needs can cause over-provisioning in certain situations, a trade-off that has not been completely examined in the current assessment.

The remainder of this report is organized as follows: Section 2-Literature Review introduces previous work on serverless platforms, cold-starts, autoscaling, and time-series forecasting. Section 3-Methodology describes data preparation, model development, and Knative integration. Section 4-Design Specification describes the design of the predictive autoscaling system. Section 5-Implementation gives detailed information about the forecasting models and how they are deployed with Knative. Section 5-Evaluation discusses the experimentation results. Section 6- Conclusion and Future Work summarizes the research and outlines future scope.

2 Related Work

2.1 Autoscaling Foundations and Serverless Cold-Start Challenges

There has been at least a decade of research on autoscaling in cloud elasticity. The initial surveys by Lorido-Botran et al. (2014) and al dhuraibi et al. (2017) have identified reactive and proactive scaling and pointed to the ongoing tradeoff between the minimization of latency and resource utilization. The reactive scaling is easy and reliable, but is incapable of handling any abrupt surges, hence reducing the quality of service. These pioneering studies defined the trade-off that has existed for a long term in the predictive autoscaling studies.

Serverless computing takes elasticity further to the function layer, at which scale to zero behaviour adds cold start delays. Lin and Glikson (2019) showed that tail latency in Knative workloads is dominated by cold starts. In the article by Tari et al. (2024), the method of autoscaling in serverless was surveyed, and it was argued that the current commercial FaaS systems use reactive triggers to activate the service, while cold starts, throttling, and performance fluctuations remained unaddressed. Efforts to model workload behaviour (including the SARIMA-based model of Jegannathan et al. (2022) only slightly reduced latency, whilst unable to predict non-stationary or strongly seasonal invocation behaviours. These preliminary surveys had however been developed at VM-level scaling, not function-level granularity. The scale-to-zero characteristics of serverless systems provide qualitatively novel challenges ie. initialization delays which can not be overcome by VM-based reactive scaling.

2.2 Adaptive, Predictive and Reinforcement Learning

Scaling, based on machine learning, has been used more in containerized solutions like Kubernetes. Machine learning methods demonstrate potential but indicate weaknesses. Mondal et al. (2023) not only attained lower latency using GRU-based prediction, but also showed poor performance in the generalization of the various types of workloads. Vu et al. (2022) used horizontal and vertical scaling, which produced 18% efficiency improvements, though with workload changes custom retraining is needed. Reinforcement learning techniques (Schuler et al. (2020); Xue et al. (2022)) are optimal trainers with

high training costs and slow convergence. Critically, the majority of research focuses on container orchestration (Kubernetes) as opposed to scale-to-zero based on serverless FaaS, which creates a niche in FaaS-based predictive autoscaling.

Hao et al. (2023) investigated continuous learning that can avoid catastrophic forgetting and incur fewer retraining expenses, and enhancing accuracy over an extended duration. An illustration of the efficiency-latency tension is proposed by Zhang et al. (2025) with archetype-based predictive autoscaling (AAPA), which minimized SLO violations but was utilizing up to 8x more resources. These approaches also show a progression of some of these strategies into adaptive learning methods, but most exist on the container orchestration layer, and fail to consider the event-driven or scale-to-zero model of FaaS providers such as Knative.

2.3 Forecasting Models for Workload Prediction

Forecast accuracy is critical in proactive autoscaling. Traditional statistical forecast models like ARIMA, SARIMA are effective in dealing with seasonality problems, yet fail to deal with abrupt bursts and invocation relationships that are non-linear in nature. Jegannathan et al. (2022) have found that SARIMA shows limited responsiveness in different serverless traffic structures.

Recent developments have gone in the direction of ML and hybrid forecasting. Guruge and Priyadarshana (2025) in their research experimented using Prophet and LSTM forecasting models on Kubernetes workloads, and the results displayed enhanced accuracy on multi-source traces. Shiekhani et al. (2025) integrated ARIMA with the use of the Random Forest with OpenWhisk and cut SLA violations by half, which confirms the possibilities of hybrid models. With a high level of accuracy, supervised regression determined the AWS demand Thota (2022), but the generalisability is quite limited because of vendor-specific restrictions. Research on cost and latency trade-offs between wider arrangements reveals that predictions can aid in cold starts in many cases that require low latency telemetry, which increases the overhead of resources (Persson and Sjöberg (2023); Golec et al. (2024); Li et al. (2025)). There is an important critical gap: forecasting in serverless autoscaling loops commonly lacks explicit cold-start mitigation. Another limitation present in most of these works is that forecasting is seldom considered directly as part of the autoscaling loop of a Serverless platform, and cold-start mitigation is not generally explicitly considered.

2.4 Balancing Scalability, Efficiency, and Runtime Constraints

Studies of resource efficiency versus scalability emphasize that both early and late provisioning can neither meet the two goals independently. Zhang et al. (2021) studied this challenge by Faster and Cheaper Serverless Computing on Harvested Resources, which presents a load-balancing mechanism that deploys FaaS workloads on ephemeral, estimated harvested VMs to minimize the cost but achieve reliability. Their experiments proved to be up to 9x throughput and 89% cheaper than normal provisioning and showed that reuse of resources could provide scale as well as efficiency to the resources in case they were managed smartly. Persson and Sjöberg (2023) were able to reduce cold starts but used timely user-navigation information, which was not practical. Golec et al. (2024) proposed a multi-objective autoscaler that used 27% of the energy and did not discuss scale-to-zero limitations. Li et al. (2025) suggested a light runtime to enhance concur-

rency in the startup of containers, which reduced boot time by 25%. Queueing delays and container initialisation are portrayed as strong determinants of scaling latency, confirmed by Govind and González-Vélez (2021) and Du et al. (2020), which is why forecasting models must be used, with a strict response budget, which is a characteristic of serverless platforms.

2.5 Summary and Research Gap

According to literature studies on Kubernetes and serverless, currently there are strategies that are responsive at the expense of efficiency, including strategies that are efficiency-oriented without preventing cold-start latency. The fact that forecasting-based scaling has demonstrated potential is evident, and recent research is seldom used to test multiple ML models on actual serverless invocation traces. Critically, none of the previous studies is a systematic comparison of many forecasting models (Prophet, LSTM, hybrid) on the scale-to-zero architecture of Knative on actual serverless traces. This paper fills that knowledge gap by incorporating Prophet, LSTM, and Hybrid forecasting models into Knatives autoscaler and studying their effectiveness on cold-start latency, responsiveness and resource efficiency given different real orchestrated workloads.

3 Methodology

3.1 Research Design

This paper takes an empirical, data-oriented design as per CRISP-DM that starts with data acquisition, then to model development, system-level assessment, and evaluation. The workflow includes the preparation of actual serverless traces, the creation of forecasting models, and their integration into the Knative autoscaling pipeline to make proactive scaling choices. The methodological steps followed in this design are summarized in Figure 1

3.2 Methodological Justification and Alternatives Considered

The methodological decisions were informed by the nature of serverless workloads that are extremely irregular, non-stationary, and cold-start sensitive. Classical statistical models (e.g., ARIMA, SARIMA) and tree-based regressors (e.g., Random Forest, XGBoost) have also been considered but were found to be inappropriate because of limited temporal awareness and their inability to adapt to sudden bursts of traffic. Architectures based on transformers were eliminated because of the computation overhead incompatibility with real-time autoscaling. Prophet, LSTM, and hybrid Prophet-LSTM were chosen as a compromise between lightweight and expressive models that can be used as microservice in Knative.

Other forms of integration strategies were also taken into consideration. Event-driven pre-warming (event driven by upstream signals) was not considered because event metadata was not present in the Azure traces. Hybrid human-in-the-loop scaling would be impractical in making high-frequency decisions, hence that was also rejected. The selected strategy that is using ML as a minimum pod floor using KPA to handle the peaks is a balance between innovation and safety, as it would enable gradual implementation without substituting well-established reactive measures.

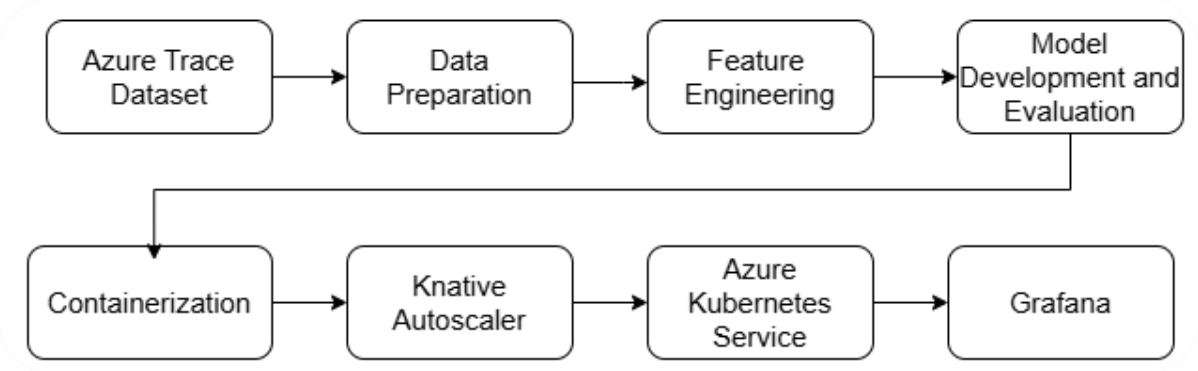

Figure 1: Overall research workflow illustrating the end-to-end process

3.3 Data Source and Preparation

The study uses the publicly available Azure Functions Invocation Trace 2021[1], which consists of timestamped invocation logs of hundreds of production serverless functions. The data was imported into Python, and manipulated to guarantee its consistency, validity and time sequence. Bad records were deleted, and timestamps were translated into associated minute-level windows to match the decision intervals of autoscaling. The resulting consolidated per-function time series of the latter was the foundation of model training and testing.

3.4 Feature Engineering

Feature engineering was carried out to transform invocation traces into inputs that could be forecasted. To describe temporal patterns, hour of day and day of week encodings were created as cyclical encodings. Indicators of idle periods and burst activity were introduced in order to capture the conditions related to cold start risks. The emphasis was on coming up with a small, generalizable feature set that can be applied to heterogeneous serverless functions without embedding workload-specific behaviours into it.

3.5 Model Development

Three forecasting models were developed- Prophet, LSTM, and Hybrid(Prophet residuals+LSTM). The long-range seasonality and the trends were picked up by Prophet. The use of LSTM was to model non-linear behaviour and short-term temporal dependence. This was combined into the hybrid method, which fitted LSTM to Prophet residuals and added up the results at inference. The individual models were trained on each of the functions with the same train-test splits to be able to compare them.

3.6 Model Evaluation

All models were tested with the same preprocessing, scaling and windowing processing on a held-out test set. Performance metrics were also calculated to evaluate the prediction accuracy and assess appropriateness for predictive autoscaling. This made sure that

[1] `https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md`

future differences in autoscaling behaviour could only be attributable to characteristics of the model and not the differences in data handling.

3.7 Knative Integration and Experiment Setup

For testing proactive autoscaling within a realistic environment, the trained models were containerised and made accessible using lightweight forecasting microservices deployed on Azure Kubernetes Service, and it was periodically queried by a custom autoscaling component which made Knative service configurations changes based on predicted demand. This arrangement facilitated comparative controlled experiments on predictive and reactive autoscaling behaviour at the same workload conditions.

3.8 Visualization

An execution monitoring workflow was deployed to store traces of execution, autoscaling decisions and model outputs to a cloud. The structured format of the logs of the experiments was exported and added to the Grafana Cloud to be visualised.

4 Design Specification

4.1 System Architecture Overview

The cloud native autoscaling pipeline is a modular system that is deployed on Azure Kubernetes Service (AKS). It aims to compare three forecasting strategies: Prophet, LSTM and Hybrid (Prophet + LSTM residuals).

Every forecasting model serves as a separate FastAPI microservice. Every model container is replicated in Azure Container Registry (ACR) and is deployed as Knative Services on AKS. An independent FastAPI-based predictive autoscaler periodically makes a request to one of such forecasting microservices, gets traffic prediction, and updates the active Knative revision with new scaling annotations. Four coordinated layers form part of the overall architecture as shown in Figure 2 (1) Forecasting Services Layer - Prophet, LSTM and Hybrid containers which operate FastAPI, (2) Autoscaler Layer - A Python script, which is an external predictive autoscaler, (3) Knative Serving Layer - Global KPA reactive autoscaler including predictive annotations on revisions, (4) Load generator, CSV metrics that are used for Grafana Cloud visualizations.

4.2 Microservices Design Forecasting

The trained models are separately deployed as Dockerized FastAPIs under each model type, and the services expose the same /predict endpoint, and hence it can be used interchangeably in the autoscaler. Prophet Service uses trend, weekly, and daily seasonality to generate predictions. LSTM Service predicts short-term sequences based on 30-step windows of the input sequence. Hybrid Service provides a combined forecast by combining the baseline of Prophet and LSTM residual forecast, which are necessary for long-term structure and local variability. The same request schema exists for all services, which enables the predictive autoscaler to run different models without altering its own logic.

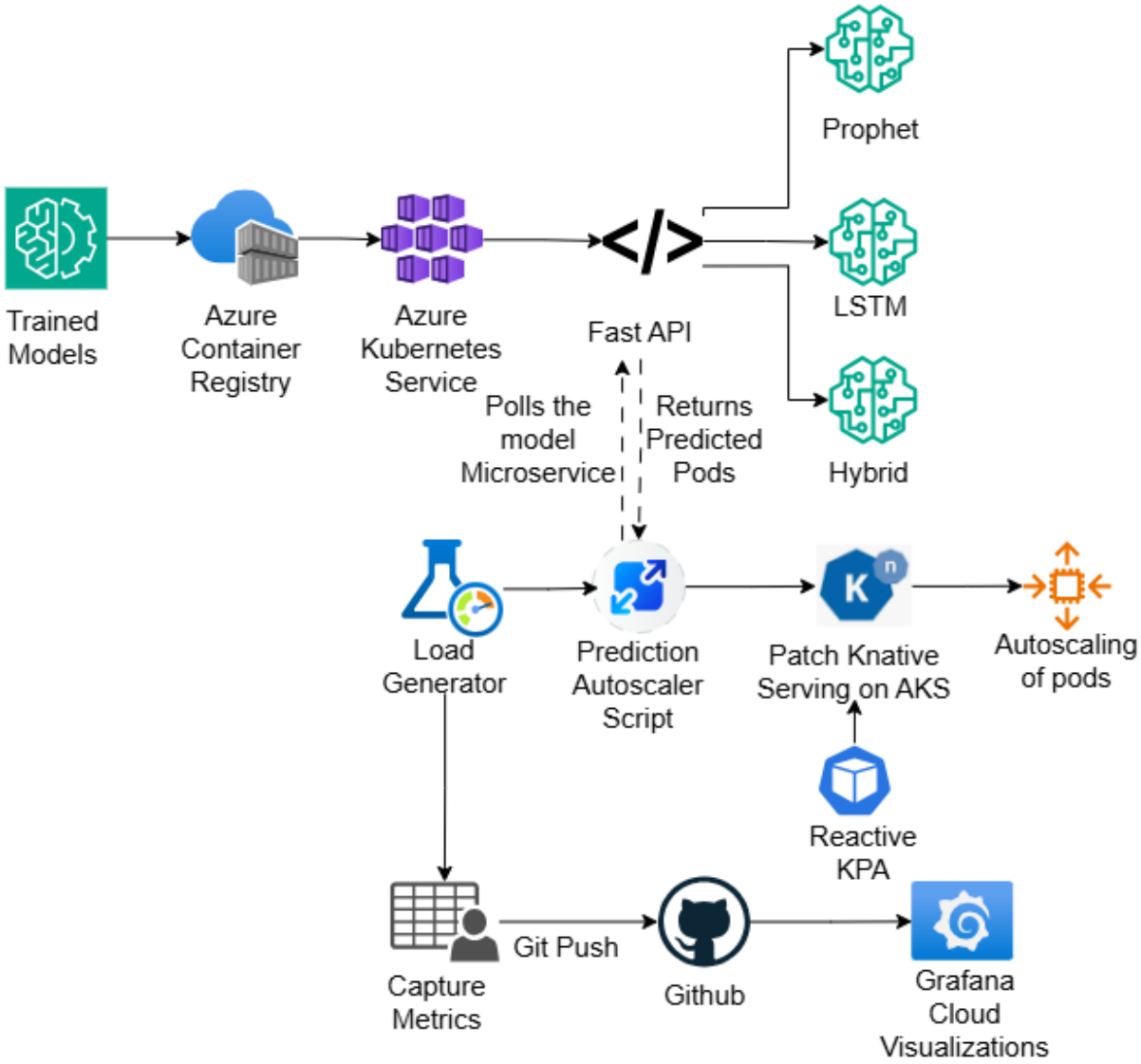

Figure 2: Architecture Diagram (Original Illustration)

The predictive autoscaler deployed as an individual Python script outside the cluster, requests predictions from the FastAPI microservice selected model, translates the acquired predictions to pod requirements based on the configured request capacity per pod, and patches the active Knative Revision by updating autoscaling.knative.dev/min-scale with the requested number of pods. Knative's built-in reactive autoscaler (KPA) is still on. The predictive autoscaler does not replace it, only that there is a set minimum floor provided by the ML model predictions, and accordingly the scale-up/scale-down occurs.

4.3 Deployment and Container Architecture

Everything is a containerized system, including three forecasting microservices, the predictive autoscaler, and the serverless functions written in Docker. Images are created using Docker on VS Code, which are then pushed to ACR. AKS pulls out images and deploys them through Knative Service manifests. Knative exposes endpoints, provided by Kourier, and are used by the autoscaler script for fetching predictions. The decoupled design helps to change forecasting models without any autoscaler modifications.

4.4 Reactive KPA Baseline

The architecture has its default Knative Pod Autoscaler (KPA), which is measured as the baseline. In the reactive-only experiments, the predictive autoscaler is not called, and KPA is only responsive to request concurrency. This is so that the four strategies, Prophet, LSTM, Hybrid, and Reactive can be tested on the same terms.

4.5 Load Generation and Metrics Capture

A custom load generator replays actual traffic traces per minute per Knative Service. It logs the number of received requests, success and error requests, p95 and p99 latency, the number of pods running and pending, cold starts, CPU and memory utilization during its execution. All measures are created locally in csv files, and they are pushed into GitHub. These CSVs are loaded in Grafana Cloud by using the Infinity data source to visualize time-series, bar charts, and XY plots.

4.6 Tools, Technologies and Experimental Environment

All the analytical processes have been implemented in Python 3.9 in Google Colab, with the help of Pandas and NumPy to prepare the data, Scikit-learn to scale the features, Prophet to forecast based on the seasonality, and TensorFlow/Keras to train the LSTM and the hybrid models. Docker Desktop was applied only to the creation and containerisation of the forecasting microservices, so the execution environment was the same for all deployments. The whole autoscaling testing, model testing, and system-level testing were only carried out using Azure Kubernetes Service (AKS), giving it an authentic cloud setup to test predictive autoscaling behaviour. Azure Container Registry (ACR) was configured to receive container images directly out of Visual Studio Code and was deployed into the Knative Serving implementation that is built on an AKS cluster. Finally, Grafana Cloud was the main dashboarding tool for visualization.

5 Implementation

The transformed time-series and engineered features, as described in Section 3, were then used to train Prophet, LSTM, and Hybrid models. The following subsections will describe the main implementation details and the artefacts produced for each model.

5.1 Prophet Model Training

The Prophet implementation began with the set of hyperparameters embodied in the experimental code. It employed a logistic growth model with the growth cap being calculated in an adaptive way as the upper percentile of training. Three forms of seasonality, which were included in the model, are multiplicative seasonality, daily and weekly pattern, and two other seasonality models, one of which is intra-day variation as modelled by a period of one day and a fourier order of 10, and the other of intra-hour variations, as modeled by a period of shorter length and a fourier order of 5. These hyperparameters enabled Prophet to model drastic change and mixed periodicity characteristic of serverless workloads. Each function was trained independently, the training experiment produced a number of outputs, and the final fitted model was saved in .pkl format for

deployment. The forecast dataframe generated by Prophet had the predicted values, and the decomposed trend and seasonal components. Prophet also created the prediction arrays over the test horizon, the test horizon error metrics (MAE, RMSE, and MAPE), and the residual series, which was to be utilized subsequently in the Hybrid design. To verify the prophet predictions with actual counts observed in the test interval, plots were visualized, which helped in ensuring that the seasonalities as well as the trend structures were fitted in the correct way before being deployed.

5.2 LSTM Model Training

The LSTM implementation took a sequence-modelling design approach, with each functional aspect creating its own neural network, which was trained on a 30-minute rolling window. The network was built up of two top to bottom stacked LSTM layers of 50 units, each with dropout in between, with a rate of 0.2 to stabilize the training process and avoid overfitting. Adam optimiser and a mean squared error loss function were used to optimise the model. The number of maximum epochs the LSTM-only model could train was set to 100, with early stopping through patience of 10 epochs, to ensure that the model would not continue training after the validation loss showed a lack of significant improvement. A 20% validation split was used to ensure the learned temporal dependencies are generalised outside of the training segment. The fitted MinMax scalers were used to scale the predictors and the targets during the implementation. The final trained network was stored to an h5 representation and a scaler object of the fitted normalisation parameters. The training process produced arrays of predicted values covering all the test horizon, the aligned timestamps, and adjusted truth values. The loss history captured the progression of the training and validation loss development, which gave confirmation that early stopping had been activated properly. Visual overlays between the LSTM predictions and the actual workload traces of every function were generated as one of the implementation outputs.

5.3 Hybrid Model Training

The Hybrid model combined the Prophet and LSTM modules in residual learning. Once the test set predictions of Prophet were obtained, a residual sequence was calculated by taking the difference between the Prophet forecast and the real observed values. This remaining series served as a training sample to a different LSTM model with the same hyperparameters to use in terms of validation split as in the LSTM-only model, such as a lookback window of 30 steps, two LSTM layers of 50 units, and dropout of 0.2 with up to 100 epochs and an early-stopping penalty of 10 epochs to prevent it from overfitting high-frequency noise. After training, the Hybrid prediction of the current timestamp was then acquired by combining the Prophet forecast with the LSTM predicted residual prediction. This step generated three implementation outputs for each function: Prophet model in pkl format, the residual LSTM in h5 format, and a scaler object that works to normalize the residual sequences. The computed accuracy measures, predicted Hybrid series, and the aligned ground truth values were stored along with the predicted Hybrid series. Combined overview of all Hybrid experiment measurements has been generated as a csv file, and Hybrid prediction curves have been drawn against Prophet and the actual trace to ensure that the combination performed as intended in all the functions.

5.4 Fast API Forecasting Services

All of the trained forecasting models were implemented through a FastAPI-based inference layer. All models, including Prophet, LSTM, and Hybrid, were packaged as isolated API services, with a single endpoint for prediction. The services preloaded their trained artefacts on startup and did all the preprocessing and postprocessing within themselves, so that the autoscaler would only get the standardized numerical forecasts. The deployment environment was constructed as a TensorFlow runtime image, which has the necessary Python dependencies, such as Prophet, TensorFlow, NumPy, Scikit-learn, and supporting libraries. Uvicorn was used to serve the services on port 8080 and packed into Docker images. Each of the containers was created using the same Dockerfile and deployed to Azure Container Registry with the same version tag.

5.5 Predictive Autoscaler Implementation and Algorithm

Algorithm 1 Predictive Autoscaling Control Loop

Initialise history buffer, model endpoint, and scaling limits
while autoscaler is running **do**
  Get the current request rate and append it to history
  Query the forecasting microservice with the recent history
  Receive prediction vector P for next k minutes
  Compute peak load: $L = \max(P)$
  Compute desired pods: $d = \lceil L/C \rceil$ where C is capacity per pod
  Clamp d within $[minScale, maxScale]$
  if d differs from current pod count **then**
    Patch Knative Revision: update `autoscaling.knative.dev/minScale = d`
  end if
  Wait for next evaluation interval
end while

The predictive autoscaler is an external control loop, which at regular intervals, requests the chosen forecasting microservice and relies on the forecasts given to make the decision about the current number of pods needed in the next interval. The autoscaler establishes a minimum value of the active Knative Revision by updating the minScale annotation according to the expected load instead of waiting for concurrency to increase. This will give a proactive floor of ready pods, but will still require the Knative Pod Autoscaler to scale further past this value as the actual traffic falls short of the prediction. Thus, the predictive autoscaler adds proactive scaling logic without conflicting with the internal scaling reactive behaviour of Knative. The general decision-making method employed in the experiments is illustrated in Algorithm 1.

5.6 Deployment of Forecasting Models in Knative

The deployment stage created a complete set of Knative service specifications that are used to execute all models. The functions had three predictive deployments, one per model, and another reactive deployment to compare the baseline. The deployments were developed as YAML manifests, which defined the container image, resource constraints,

environment variables, and scaling annotations. After implementing on the Azure Kubernetes Service, Knative automatically obtained external URLs per service with the help of Kourier and SSLIP routing. All images were retrieved from Azure Container Registry by the version tag that was built beforehand. Finally, all forecasting strategies were tested individually against the same workload trace.

5.7 Load Generation and Experiment Logs

A custom load generator replayed invocation traces at Knative endpoints. Per-minute metrics like requests sent /successful/ errors, latency (average, p50, p95, p99, max), running/ pending pods, CPU usage, memory usage and cold start events. Finally, logs were exported in the form of structured CSVs for analysis.

5.8 Visualisation and Monitoring in Grafana Cloud

The CSV experiment logs were pushed to GitHub and consumed by the Grafana Cloud using the infinity data source. Dashboards that were developed in the process of the implementation consisted of a time-series of invocation behaviour, pod activity, distributions of latency, CPU and memory usage, and cold-start events. Other XY charts were employed in investigating relationships like the latency versus pod count in the various models. The visualizations were used to monitor the autoscaling behaviour in the experiments and to ensure that the autoscaler and forecasting services were working as planned, thus it formed the basis of comparative evaluation.

5.9 Reproducibility

All the code, configuration files, and experiment scripts required to reproduce the forecasting models, data preparation workflow, and Knative autoscaling experiments are made available in a public GitHub repository[2] to make sure that the study is reproducible. The repository contains the complete Fast API microservice implementations of Prophet, LSTM and Hybrid models, Docker build artifacts, Kubernetes and Knative deployment artifacts, and the Python-based predictive autoscaler.

6 Evaluation

This chapter evaluates the predictive autoscaling solution in two phases: offline training of models and online Knative deployment. The analysis is oriented on the behaviour of Prophet, LSTM, and Hybrid models in the process of forecasting, and how these behaviours affected the actual results of autoscaling of three typical Azure Functions.

6.1 Evaluation of Model Training Results

The training outcomes have shown a strong relationship between the workload structure and the model behaviour. Prophet converged effectively to all eight functions, however, the predictions differed vastly based on the temporal patterns. For some functions, Prophet collapsed to an almost zero trend in the test window. Table 1 summarizes the

[2] `https://github.com/abigail-anil/knative-predictive-autoscaler`

training accuracy of the three forecasting models for the selected functions. Prophet didn't perform well for highly bursty functions like func110 and func235. This behaviour is an indication of the assumptions made in the model: Prophet breaks down a time series into seasonality and trend, and both of them are weak or non-existent in burst-dominated serverless traffic.

Table 1: Training Performance of Prophet, LSTM and Hybrid Models (Selected Functions)

Function	**Prophet**		**LSTM**		**Hybrid**	
	MAE	**RMSE**	**MAE**	**RMSE**	**MAE**	**RMSE**
`func_235`	25.89	36.32	10.56	21.09	11.43	22.13
`func_126`	28.26	38.21	1.94	6.15	2.16	6.11
`func_110`	2.54	28.64	4.63	27.81	5.17	27.57

Prophet acted more sensibly for heavy and steady workloads like func126, where the model can resolve a consistent seasonal addressing. In such instances, Prophet generated smooth long-range predictions and had an average accuracy. Such a difference aligns with the literature sources, where Prophet is repeatedly reported to operate best when seasonality is high and worse when the data is irregular or full of spikes. The LSTM model presented a significantly high predictive accuracy at each workload. It had a better capacity to track the increasing and decreasing activity as compared to Prophet because it was able to learn local temporal relationships. Although the test sets had sudden bursts, LSTM was able to pick the overall structure, but again, extreme bursts were sometimes under-predicted. These constraints frequently occur in sequence models trained on very small lookback windows, yet the overall behaviour was much higher than Prophet on burst-driven traces. The performance of the Hybrid model relied on the output of Prophet. In workloads where Prophet was generating any important signal-to-noise predictive variation, like in func126, Hybrid made the most predictable and precise forecasts by taking all the long-term structure of the predictor of Prophet and making all the short-term corrections of LSTM. In burst heavy workloads, the Hybrid output was also similar to LSTM since Prophet does not add a lot of usable information. This is also true of the hybrid residual theory: hybridization is only value adding in cases where there is complementary and independent dynamics encoded in both components. Collectively, the outcomes of training validate the idea that there is no model that is best in every situation. Prophet suits the steady workloads, LSTM fits all types of workloads, and Hybrid fits better in a workload that has both steady, trendy workload and temporary variation.

6.2 Runtime Evaluation in Knative

6.2.1 Func235 (periodic batch workload with long sustained traffic blocks)

The most convincing illustration of the enhancement of behaviour by predictive autoscaling was Func_235. The workload has long and steady blocks of traffic separated by abrupt zero drops. After the warming up of all blocks, all of the strategies stabilised at a similar low p95 latency. As shown in Figure 3, Prophet followed the reactive base especially in these steady conditions, where predictions paralleled the actual pattern when the

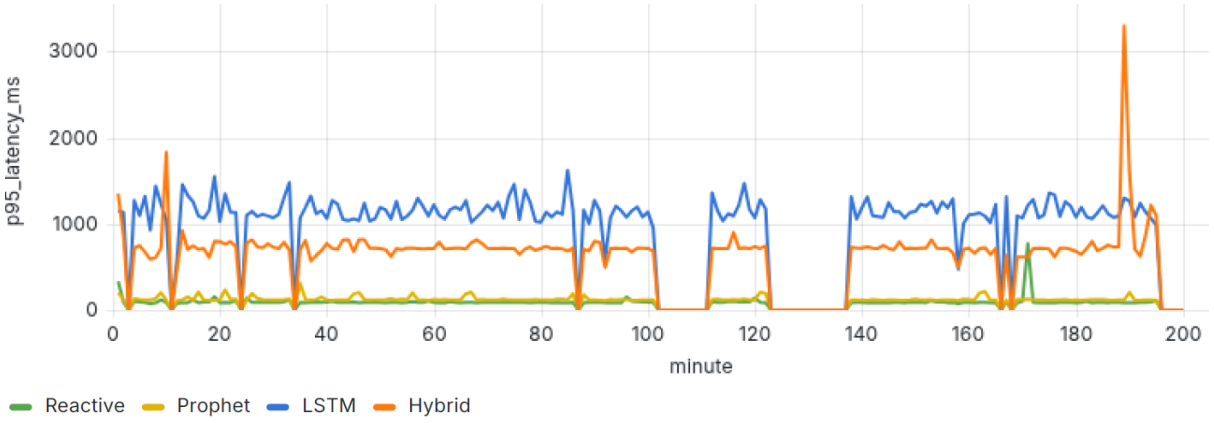

Figure 3: Latency for periodic batch workload with long sustained traffic blocks

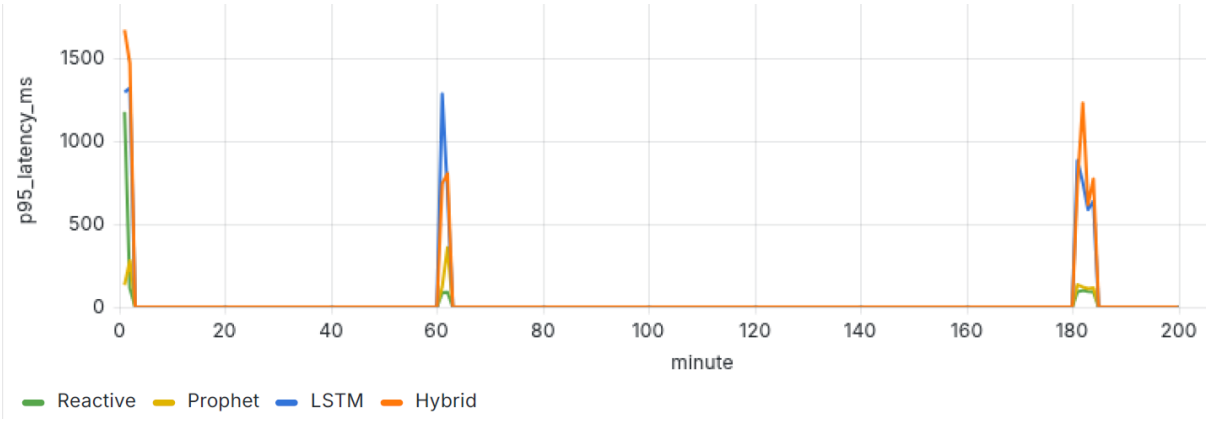

Figure 4: Model-wise Latency Comparison for sparse and highly bursty workload

workload was periodic. The significant differences were revealed only during transitions of idle periods. Reactive scaling always experienced cold-start spikes, whereby it does not react until the concurrency increases. These spikes were greatly mitigated by all the predictive strategies, which is by pre-scaling in advance. Hybrid was marginally higher than Prophet but lower than LSTM among the predictive models; it was during most blocks that LSTM had the highest latency, and the cost of its model-loading and inference was heavier than the other two. Func_235 affirms the fact that predictive autoscaling is most advantageous in workloads whose transitions are more important than workload steady-state behaviour.

6.2.2 Func110 - Sparse and Highly Bursty Workload

Func 110 has very short bursts with big periods of idle time. Since the spikes are too short to materialise into full cold-start penalties, reactive and Prophet also obtain a similar low latency as shown in Figure 4. Predictive autoscaling, even during this load level, exhibits steady and consistent behaviour in comparison to reactive autoscaling. which sometimes experiences jitter during warm-up. The higher peaks of LSTM and Hybrid are explained by the model-loading costs, but the predictive strategies still do not have the worst-case cold-start behaviour as would manifest itself in case bursts were longer or more frequent. This workload is thus a lower bound, even in the situations where prediction benefits do not attain their full potential, predictive autoscaling does not increase or stabilize but

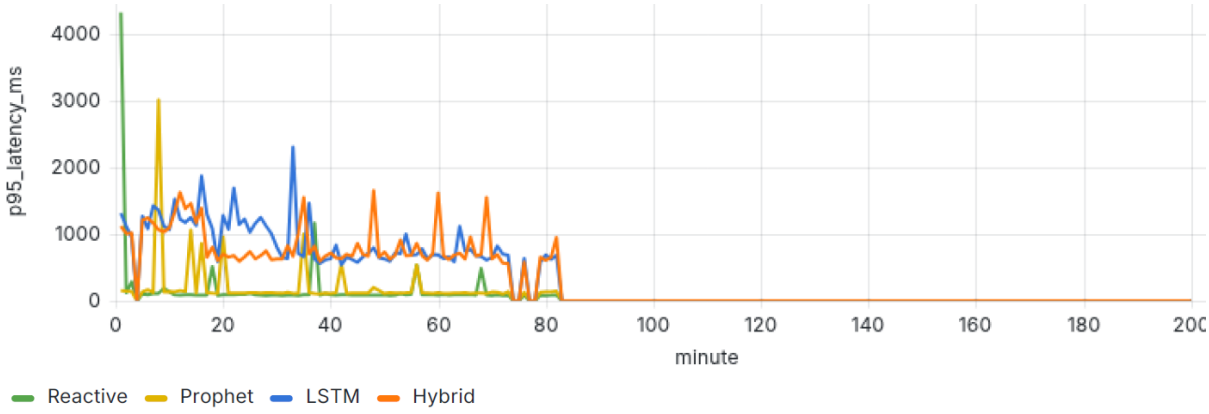

Figure 5: Model-wise Latency Comparison for steady high workload

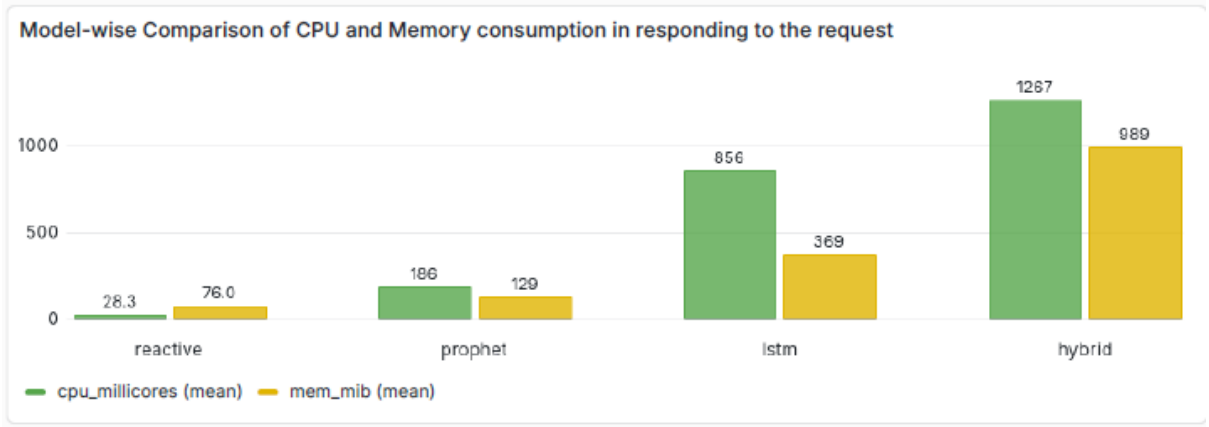

Figure 6: Model-wise comparison of average CPU and memory usage during autoscaling.

never decreases in performance.

6.2.3 Func126 - Steady-High Workload

As shown in Figure 5, Func126 is not a sharp fluctuator and is steady and active at all times. Both LSTM and Hybrid had shown great error reductions in training, indicating high learnability. In Knative, the image is different: when initially cold starts settle, both reactive and predictive strategies have almost identical latency, but reactive sometimes seems to be lower simply because it does not even need to load a model. Models need to be loaded through predictive routes, compute forecasts, and scaling signals, and this presents an inconvenient but inevitable overhead to workloads that do not need any anticipation. Even in this case, predictive autoscaling is not at a disadvantage. Prophet and Hybrid are highly related to tracking reactive, indicating that predictive control is stable and safe when its advantages are not glaring. What is more important is that the bursts are absent and that there is just no future spike to pre-scale. The important conclusion is that predictive autoscaling is as good as reactive in the steady workloads and still has the capacity to be better than that in burst-heavy applications.

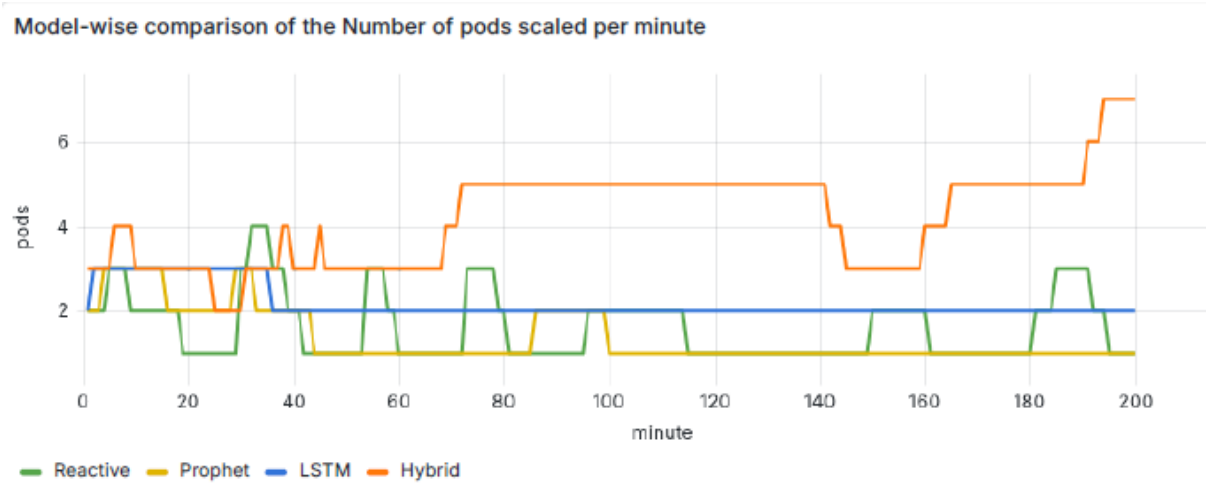

Figure 7: Model-wise pod-scaling timeline, highlighting how predictive autoscalers pre-scale earlier and reduce jitter relative to the reactive KPA.

6.2.4 CPU and Memory Behaviour

CPU and memory utilization of each of the running pods from the CSV logs helped to analyze the resource implications of running predictive autoscaling. As also shown in Figure 6, reactive control always had the lowest footprint since it does not carry out model inference. Prophet used small amount of CPU during predictions, and the memory space needed was also minimal. LSTM and Hybrid showed short-lived CPU and memory spikes whenever a scale-out event was triggered, causing fresh model loading and notably cold starts. These spikes faded away after the pods became warm, and all the strategies then developed extremely similar behaviour in terms of resource use. This means that predictive autoscaling only introduces overhead when a forecast is generated and not when it is implemented in steady mode.

6.2.5 Scaling Smoothness During Transitions

The scaling traces depict that predictive autoscaling provides smoother transitions in the event of a load change. Reactive scaling only acts once concurrency has increased, and thus it takes time to start responding, causing jitters in warm-up. In comparison, all predictive models begin creating pods a little sooner, resulting in more controlled transitions with less sudden shifts. Prophet was less likely to respond quickly to burst heavy workloads, whereas LSTM and Hybrid were quick in their response when the traffic pattern showed spikes. As Figure 7 demonstrates, predictive and reactive behaviour became graphically identical (when the system was stabilized), which shows that anticipating load never destabilized Knative's internal scaling loop.

6.3 Discussion - Synthesis and Interpretation

In all three workload types, there is a distinct pattern that appears. Predictive autoscaling is not supposed to replace reactive control, rather extend it. Prediction can be used to mitigate latency in cases where reactive scaling fails to respond to workloads that are time sensitive, e.g., bursty traffic or sudden change because reactive decisions do not occur until the load is detected. For constant and predictable workloads, the predictive

strategy is as predictable as the reactive one, and there is very little overhead due to loading models. That is, it never has a negative effect on performance in periods of calmness and always has a positive effect when the traffic gets irregular, which is directly linked to the research question on the reduction of cold-start and readiness to bursts.

Conservative pre-scaling technique places less emphasis on cost minimization and emphasizes availability, which tolerates some amount of temporary over-provisioning to minimize cold starts. During workload transitions, predictive scaling will keep extra warm capacity in advance of demand, improving latency responsiveness and potentially consuming more resources than purely reactive scaling. This also brings out the trade-off between cost-responsiveness and predictive autoscaling: predictive autoscaling obtains the benefit of cold-start delays at the expense of preparedness (idle pods within pre-scale windows). The reactive scaling can be cheaper to use in workloads where latency does not matter, but latency-sensitive services can be worth this overhead to ensure that the SLOs are met.

The use of resources goes the same way. Predictive autoscaling is a little more CPU and memory-intensive in terms of transitions since forecasting and pre-scaling are in advance of the load. This leads to smooth scaling and reduced cold start delays. After the system stabilizes, Prophet and Hybrid both arrive at an identical value of the pods as reactive and demonstrate that prediction results do not lead to sustained over-provisioning once workload stabilizes. LSTM has higher inference overhead, but still provides better responsiveness, whereas Hybrid is the most balanced among the three. Overall, the resource trade-off is a small one relative to the benefit of reliability acquired during work load shifts.

In this study, explicit cost modelling was not carried out. The patterns of observed resources, however, show that predictive strategies can make more use out of the scaling transitions, which implies that there may be a cost overhead of keeping warm instances. Related work has indicated that the existence of more aggressive predictive autoscalers can result in heavy overprovisioning in exchange for eliminating SLO violations (e.g., Zhang et al., 2025). Future work should quantify this trade-off using provider-specific pricing models to assess when latency benefits outweigh additional compute cost.

One of the final benefits of this work is that it does not compare Prophet, or LSTM, or Hybrid models in isolation, but implements them directly within a real Knative autoscaling controller. To the best of my knowledge, this is one of the few end-to-end implementations of prediction-driven autoscaling within Knative by testing scaling behaviour with live workloads using real invocation traces. The findings indicate the behaviour of each of the models. Hybrid works best on volatile or semi-structured traces, LSTM works best on average noisy workloads and Prophet is not able to survive burst patterns. Assessment also brings out the effect of model complexity, load time, and functionality-specific behaviour on real scaling performance, which is not discussed in other previous works.

The other lesson learned is the trade-off of selecting a scaling strategy. Prediction adds model loading and inference overhead, which leads to a slight penalty on latency as compared to reactive decisions. However, it is usually useful in applications involving pipelines like streaming ingestion systems, anomaly detection, or periodic analytics with a less significant priority of latency per request than readiness. In case the APIs are ultra-latency-sensitive, then reactive control still might be the safer option.

The study is reinforced by two factors- full-cycle testing with actual Knative deployments using production-based invocation traces and contrasting across three types of

workloads that indicate clearly where predictive autoscaling actually serves an important purpose. There are still several limitations, such as the overhead of loading models with larger models, the instability of Prophet with unstable traces that is transmitted to the Hybrid model and restricted workloads. Nonetheless, this does not alter the key point that predictive autoscaling is needed most in precisely the factual conditions that reactive approaches are insufficient.

7 Conclusion and Future Work

The research investigated the issue of whether machine learning based predictions could enhance autoscaling in serverless computing by minimizing cold start delays and transition latency. Prophet, LSTM, and Hybrid models were trained using real Azure Functions traces and integrated into Knative, which makes scaling decisions based on predictions made by the model. The experiments revealed that LSTM and Hybrid models provided the most reliable predictions and tended to scale responsively more in cases of bursty and transition-heavy workload, and with Prophet, it was only reliable with steady and seasonal traces. Notably, predictive autoscaling did not degrade performance in steady workloads and had no destabilizing impact on the Knative control loop, which suggests that the use of ML models for forecasting is effective over reactive control. One of the contributions made by this work is end-to-end assessment of forecasting models within a real autoscaling pipeline that demonstrated how the offline predictive performance can be transformed into scalability behaviour in a production like setup. The limitation of this work is that there is no explicit cost analysis. Although the use of resources was measured, the cost-performance trade-offs were not quantified. Conservative pre-scaling strategy likely adds costs associated with transition periods due to temporary overprovisioning, as a trade-off, which may be better explored using provider-specific pricing models.

Future scope of research would be to consider ways to minimise model loading overhead by using warm model containers, on-node caching or simpler forecasting architectures to support fast inference. The combination of predictive signals and adaptive mechanisms like reinforcement learning based autoscalers might be essential in enhancing responsiveness in situations with very irregular traffic. Generalizability would also be improved by extending the evaluation to multi-tenant clusters, the use of GPU accelerated serverless workloads or a greater number of heterogeneous functions. Lastly, exploring uncertainty-aware predictions or dynamic reconfiguration of models at runtime can be used to optimize proactive scaling policies of production-scale serverless systems.

References

al dhuraibi, Y., Fawaz, P., Djarallah, N. and Merle, P. (2017). Elasticity in cloud computing: State of the art and research challenges, *IEEE Transactions on Services Computing* **PP**: 1–1. DOI: 10.1109/TSC.2017.2711009, Scopus Q1, CiteScore 10.8, SJR 1.441. **URL:** *https://ieeexplore.ieee.org/document/7937885*

Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q. and Chen, H. (2020). Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, p. 467–481. DOI: 10.1145/3373376.3378512.
URL: *https://doi.org/10.1145/3373376.3378512*

Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S. and Uhlig, S. (2024). Cold start latency in serverless computing: A systematic review, taxonomy, and future directions, *ACM Comput. Surv.* **57**(3). DOI: 10.1145/3700875, Scopus Q1, CiteScore 51.6, SJR 5.797.
URL: *https://doi.org/10.1145/3700875*

Govind, H. and González-Vélez, H. (2021). Benchmarking serverless workloads on kubernetes, *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* pp. 704–712. DOI: 10.1109/CCGrid51090.2021.00085.
URL: *https://api.semanticscholar.org/CorpusID:236920433*

Guruge, P. and Priyadarshana, Y. (2025). Time series forecasting-based kubernetes autoscaling using facebook prophet and long short-term memory, *Frontiers in Computer Science* **7**. DOI: 10.3389/fcomp.2025.1509165.
URL: *https://doi.org/10.3389/fcomp.2025.1509165*

Hao, H., Chu, Z., Zhu, S., Jiang, G., Wang, Y., Jiang, C., Zhang, J. Y., Jiang, W., Xue, S. and Zhou, J. (2023). Continual learning in predictive autoscaling, *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, CIKM '23, Association for Computing Machinery, New York, NY, USA, p. 4616–4622. DOI:10.1145/3583780.3615463, CORE Rank: A, Cited by 5.
URL: *https://doi.org/10.1145/3583780.3615463*

Hong, S., Kim, Y., Nam, J. and Kim, S. (2024). On the analysis of inter-relationship between auto-scaling policy and qos of faas workloads, *Sensors* **24**(12). DOI: 10.3390/s24123774, Scopus Q1, CiteScore 8.2, SJR 0.764.
URL: *https://www.mdpi.com/1424-8220/24/12/3774*

Jegannathan, A. P., Saha, R. and Addya, S. K. (2022). A time series forecasting approach to minimize cold start time in cloud-serverless platform, *2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pp. 325–330. DOI: 10.1109/BlackSeaCom54372.2022.9858271, Cited by 31.
URL: *https://ieeexplore.ieee.org/document/9858271*

Lee, S., Yoon, D., Yeo, S. and Oh, S. (2021). Mitigating cold start problem in serverless computing with function fusion, *Sensors* **21**(24). DOI: 10.3390/s21248416, Scopus Q1, CiteScore 8.2, SJR 0.764.
URL: *https://www.mdpi.com/1424-8220/21/24/8416*

Li, Z., Wu, C., Xu, C., Chen, Q., Quan, S., Zha, B., Wang, Q., Han, W., Wu, J. and Guo, M. (2025). Lightweight and holistic-scalable serverless secure container runtime for high-density deployment and high-concurrency startup, *IEEE Transactions on Computers* **74**(8): 2621–2634.
URL: *https://ieeexplore.ieee.org/document/11008773*

Lin, P. and Glikson, A. (2019). Mitigating cold starts in serverless platforms: A pool-based approach, *CoRR* **abs/1903.12221**. DOI: 10.48550/arXiv.1903.12221.
URL: *http://arxiv.org/abs/1903.12221*

Lorido-Botran, T., Miguel-Alonso, J. and Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments, *Journal of Grid Computing* **12**(4): 559–592. DOI: 10.1007/s10723-014-9314-7.
URL: *https://doi.org/10.1007/s10723-014-9314-7*

Mondal, S., Wu, X., Kabir, H. M. D., Dai, H.-N., Ni, K., Yuan, H. and Wang, T. (2023). Toward optimal load prediction and customizable autoscaling scheme for kubernetes, *Mathematics* **11**: 2675. DOI: 10.3390/math11122675, Scopus Q1, CiteScore 4.6, SJR 0.498.
URL: *https://www.mdpi.com/2227-7390/11/12/2675*

Persson, G. and Sjöberg, W. (2023). *Mitigating serverless cold starts through predicting computational resource demand: Predicting function invocations based on real-time user navigation*, PhD thesis. DOI: 10.13140/RG.2.2.27846.04168.

Schuler, L., Jamil, S. and Kühl, N. (2020). Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. DOI: 10.48550/arXiv.2005.14410, Cited by 15.
URL: *https://www.computer.org/csdl/proceedings-article/ccgrid/2021/958600a804/1vK0sY2LDtS*

Shiekhani, L., Wang, H., Shi, W., Liu, J., Qiu, Y., Gu, C. and Ding, W. (2025). The hybrid model: Prediction-based scheduling and efficient resource management in a serverless environment, *Applied Sciences* **15**. DOI: 10.3390/app15147632.
URL: *https://doi.org/10.3390/app15147632*

Tari, M., Ghobaei-Arani, M., Pouramini, J. and Ghorbian, M. (2024). Auto-scaling mechanisms in serverless computing: A comprehensive review, *Computer Science Review* **53**: 100650. DOI: https://doi.org/10.1016/j.cosrev.2024.100650, Scopus Q1, CiteScore 38.4, SJR 3.276, Cited by 43.
URL: *https://www.sciencedirect.com/science/article/pii/S1574013724000340*

Thota, R. C. (2022). Intelligent auto-scaling in aws: Machine learning approaches for predictive resource allocation, *International Journal of Scientific Research and Management (IJSRM)* **10**: 1–8. DOI: 10.18535/ijsrm/v10i10.ec06, Cited by 14.

Tràn, M.-N. and Kim, Y. (2023). Optimized resource usage with hybrid auto-scaling system for knative serverless edge computing, *Future Generation Computer Systems* **152**: 304–316. DOI: 10.1016/j.future.2023.11.010.
URL: *https://doi.org/10.1016/j.future.2023.11.010*

Vu, D.-D., Tran, M.-N. and Kim, Y. (2022). Predictive hybrid autoscaling for containerized applications, *IEEE Access* **10**: 109768–109778. DOI: 10.1109/ACCESS.2022.3214985, Scopus Q2, CiteScore 9.0, SJR 0.849, Cited by 38.
URL: *https://ieeexplore.ieee.org/document/9919851*

Xue, S., Qu, C., Shi, X., Liao, C., Zhu, S., Tan, X., Ma, L., Wang, S., Wang, S., Hu, Y., Lei, L., Zheng, Y., Li, J. and Zhang, J. (2022). A meta reinforcement learning approach for predictive autoscaling in the cloud. DOI: 10.1145/3534678.3539063, CORE Rank: A*, Cited by 52.
URL: *https://doi.org/10.1145/3534678.3539063*

Zhang, G., Vippagunta, S., Nandagopal, R., Raman, S., Xu, J., Pfeiffer, M., Chatterjee, S., Tan, Z., Guo, W. and Jiang, H. (2025). Archetype-aware predictive autoscaling with uncertainty quantification for serverless workloads on kubernetes.
URL: *https://doi.org/10.48550/arXiv.2507.05653*

Zhang, Y., Goiri, I. n., Chaudhry, G. I., Fonseca, R., Elnikety, S., Delimitrou, C. and Bianchini, R. (2021). Faster and cheaper serverless computing on harvested resources, *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, Association for Computing Machinery, New York, NY, USA, p. 724–739. DOI: 10.1145/3477132.3483580.
URL: *https://doi.org/10.1145/3477132.3483580*