# **Ανάπτυξη ML Pipeline για την Πρόβλεψη Χρήσης Πόρων σε Kubernetes**

**Φοιτητές:** Νικόλαος Τσιούτσιας - ΑΜ: 1072479  
 Ανδρέας Ιωαννίδης - ΑΜ: 1093373

## **Περίληψη**

Η παρούσα εργασία επικεντρώνεται στην ανάπτυξη ενός ολοκληρωμένου ML pipeline για την πρόβλεψη της χρήσης πόρων (CPU και μνήμης) σε περιβάλλον Kubernetes. Το σύστημα που υλοποιήθηκε συλλέγει δεδομένα χρονοσειρών από το Prometheus, εκπαιδεύει μοντέλα μηχανικής μάθησης με XGBoost, και αναπτύσσει τα εκπαιδευμένα μοντέλα ως microservices στο Kubernetes cluster. Λόγω τεχνικών προκλήσεων, η υλοποίηση έγινε με Python scripts αντί για Kubeflow, διατηρώντας ωστόσο την πλήρη λειτουργικότητα του συστήματος πρόβλεψης.

## **1. Εισαγωγή**

### **1.1 Στόχος της Εργασίας**

Ο κύριος στόχος της εργασίας είναι η δημιουργία ενός αυτοματοποιημένου συστήματος που μπορεί να προβλέπει τη μελλοντική χρήση πόρων (CPU και μνήμης) σε ένα Kubernetes cluster, επιτρέποντας έτσι την προληπτική κλιμάκωση και βελτιστοποίηση των πόρων.

### **1.2 Τεχνολογίες που Χρησιμοποιήθηκαν**

* **Kubernetes**: Ως πλατφόρμα ορχήστρωσης containers
* **Prometheus**: Για τη συλλογή μετρικών
* **Grafana**: Για την οπτικοποίηση των μετρήσεων
* **Python**: Για την ανάπτυξη του ML pipeline
* **XGBoost**: Ως αλγόριθμος μηχανικής μάθησης
* **FastAPI**: Για τη δημιουργία REST API endpoints
* **Docker**: Για το containerization των μοντέλων

## 

## **2. Αρχιτεκτονική Συστήματος**

## 

### **2.1 Επισκόπηση Αρχιτεκτονικής**

Το σύστημα αποτελείται από τα εξής βασικά στοιχεία:

1. **Kubernetes Cluster**: Το κεντρικό περιβάλλον εκτέλεσης.
2. **Web app:** Μία απλή εφαρμογή η οποία επιστρέφει ένα MB σε κάθε HTTP request στο endpoint “/download” και επιστρέφει ένα Hello στο root endpoint “/”.
3. **Prometheus Server**: Συλλέγει μετρικές από τα pods, αποθηκεύει τα δεδομένα χρονοσειρών.
4. **Grafana**: Αποτυπώνει πολλαπλά δεδομένα με χρήση ενός dashboard.
5. **ML Pipeline**: Επεξεργάζεται δεδομένα και εκπαιδεύει μοντέλα.
6. **Prediction Services**: Microservices για real-time προβλέψεις.

### **2.2 Ροή Δεδομένων**

Τα δεδομένα δημιουργούνται μέσω της παραγωγής traffic από το LOCUST script. Έπειτα, ο cAdvisor παρέχει πληροφορίες για την επικοινωνία και τη λειτουργία των containers μέσα στο cluster, ενώ ο Prometheus συλλέγει και αποθηκεύει αυτά τα δεδομένα. Τέλος, κατεβάζουμε csv αρχεία από το Grafana, για memory και cpu και τα δίνουμε στο αντίστοιχο pipeline για την παραγωγή του μοντέλου πρόβλεψης.

## 

## **3. Υλοποίηση**

### **3.1 Διαμόρφωση Περιβάλλοντος**

Αρχικά, δημιουργήσαμε ένα Kubernetes cluster και εγκαταστήσαμε τα απαραίτητα εργαλεία. Ακολουθεί παράδειγμα (για περισσότερες λεπτομέρειες δείτε το commands script που δίνεται):

**//** HELM PROMETHEUS INSTALL AND GRAFANA

**helm** repo add prometheus-community https://prometheus-community.github.io/helm-charts

**helm** repo update

**helm** install prometheus-operator prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace

**kubectl** get pods -n monitoring

**kubectl** port-forward -n monitoring svc/prometheus-operator-prometheus 9090

**kubectl** port-forward -n monitoring svc/prometheus-operator-grafana 3000

### **3.2 Δημιουργία Φορτίου και Προσομοίωση Κίνησης**

Για τη δημιουργία ρεαλιστικού φορτίου, αναπτύξαμε μια εφαρμογή web server που δέχεται HTTP αιτήματα. Χρησιμοποιήσαμε εργαλεία όπως το LOCUST για την προσομοίωση κίνησης με μεταβλητό φορτίο κατά τη διάρκεια 24 ωρών (locust.py).

### **3.3 Συλλογή Δεδομένων**

Τα δεδομένα συλλέχθηκαν από το Prometheus χρησιμοποιώντας τα παρακάτω queries:

**CPU Usage:**

rate(container\_cpu\_usage\_seconds\_total{namespace="app", pod=~"my-app-deployment.\*"}[1m])

Υπολογίζει τον μέσο ρυθμό κατανάλωσης CPU (σε core-seconds ανά δευτερόλεπτο – δηλαδή, πόσο χρόνο CPU χρησιμοποιείται ανά δευτερόλεπτο πραγματικού χρόνου), για containers που εκτελούνται σε pods του deployment my-app-deployment στο namespace app, με βάση τα δεδομένα του τελευταίου 1 λεπτού.

**Memory Usage:**

container\_memory\_usage\_bytes{namespace="app", pod=~"my-app-deployment.\*"}

Επιστρέφει την τρέχουσα χρήση μνήμης (σε bytes) από τα containers που εκτελούνται σε pods του deployment my-app-deployment, στο namespace app. Το metric αυτό αποτυπώνει σε πραγματικό χρόνο τη μνήμη RAM που έχει δεσμεύσει κάθε container.

Συλλέξαμε δεδομένα διάρκειας 24 ωρών μέσω του cAdvisor, ο οποίος παρέχει λεπτομερή στατιστικά σε επίπεδο container, όπως χρήση CPU, μνήμης, δίσκου και δικτύου. Η συλλογή έγινε στο namespaceapp, αξιοποιώντας τη δυνατότητα built-in service discovery του Prometheus.

Συγκεκριμένα, ο Prometheus εντόπισε αυτόματα τα pods που εκτελούνται στο cluster μέσω της Kubernetes service discovery μηχανής, η οποία βασίζεται σε labels, annotations και namespaces για να αναγνωρίσει targets. Έτσι, χωρίς χειροκίνητη καταγραφή των endpoints, ο Prometheus κατάφερε να ανακαλύψει και να παρακολουθήσει δυναμικά τα containers του app namespace, ακόμα και όταν δημιουργήθηκαν ή διαγράφηκαν κατά τη διάρκεια της περιόδου συλλογής.

### 

### **3.4 Ανάπτυξη ML Pipeline**

#### **3.4.1 Προεπεξεργασία Δεδομένων**

Η προεπεξεργασία περιλάμβανε:

* Καθαρισμό των δεδομένων από outliers
* Κανονικοποίηση των τιμών
* Δημιουργία lagged features για τη μοντελοποίηση χρονοσειρών

def add\_lags(df, target\_col, lags=15):

df\_lagged = df.copy()

for i in range(1, lags + 1):

df\_lagged[f'{target\_col}\_lag\_{i}'] = df\_lagged[target\_col].shift(i)

df\_lagged = df\_lagged.dropna().reset\_index(drop=True)

return df\_lagged

#### **3.4.2 Εκπαίδευση Μοντέλου**

## **3.4.2.1 Επιλογή Αλγορίθμων και Στρατηγική Προσέγγισης**

Η επιλογή των κατάλληλων αλγορίθμων για την πρόβλεψη των μετρικών CPU και Memory αποτέλεσε κρίσιμη απόφαση που βασίστηκε στη φύση και τα χαρακτηριστικά κάθε μετρικής. Μετά από προκαταρκτική ανάλυση των δεδομένων, διαπιστώθηκε ότι οι δύο μετρικές παρουσιάζουν εντελώς διαφορετική συμπεριφορά, γεγονός που οδήγησε στην υιοθέτηση διαφορετικών προσεγγίσεων για το καθένα.

Για την πρόβλεψη της χρήσης CPU επιλέχθηκε ο αλγόριθμος XGBoost Regressor, καθώς η μετρική αυτή εμφανίζει έντονη μεταβλητότητα και μη-γραμμικές σχέσεις που απαιτούν ισχυρό ensemble learning αλγόριθμο. Το XGBoost έχει αποδειχθεί ιδιαίτερα αποτελεσματικό σε χρονοσειρές με υψηλό θόρυβο και πολύπλοκα μοτίβα. Το μοντέλο παραμετροποιήθηκε με 100 decision trees και learning rate 0.1, ενώ χρησιμοποιήθηκαν τα προηγούμενα 15 σημεία (30 λεπτά ιστορικών δεδομένων) ως χαρακτηριστικά εισόδου.

Αντίθετα, για την πρόβλεψη της χρήσης μνήμης υιοθετήθηκε μια πιο απλή προσέγγιση με Linear Regression. Η επιλογή αυτή δικαιολογείται από το γεγονός ότι η χρήση μνήμης των containerized εφαρμογών τείνει να παρουσιάζει πιο γραμμικές τάσεις και σταθερότερη συμπεριφορά με λιγότερες απότομες διακυμάνσεις. Πριν την εκπαίδευση, εφαρμόστηκε εξομάλυνση των δεδομένων μέσω κυλιόμενου μέσου όρου με παράθυρο 5 τιμών για την αφαίρεση του στατιστικού θορύβου, ενώ χρησιμοποιήθηκαν 8 προηγούμενα σημεία (16 λεπτά) ως lag features.

**3.4.2.2 Επεξεργασία Δεδομένων**

Ένα από τα πιο σημαντικά στοιχεία της προετοιμασίας ήταν η δημιουργία των lag features, που αποτελούν τη βάση για την κατανόηση των χρονικών εξαρτήσεων. Κάθε μοντέλο εκπαιδεύτηκε να "κοιτάζει" πίσω σε ένα συγκεκριμένο παράθυρο ιστορικών δεδομένων για να προβλέψει την επόμενη τιμή. Η επιλογή του μεγέθους αυτού του παραθύρου έγινε βάσει πειραματισμού και κατανόησης των χαρακτηριστικών κάθε μετρικής.

## **3.4.2.3 Στρατηγική Εκπαίδευσης και Διαχωρισμός Δεδομένων**

Η στρατηγική διαχωρισμού των δεδομένων σχεδιάστηκε για να προσομοιώνει ρεαλιστικά συνθήκες παραγωγής. Για το CPU μοντέλο, τα πρώτα 576 σημεία (19.2 ώρες) χρησιμοποιήθηκαν για εκπαίδευση, ενώ τα τελευταία 144 σημεία (4.8 ώρες) κρατήθηκαν για αξιολόγηση. Αυτή η αναλογία 80-20 εξασφαλίζει επαρκή δεδομένα εκπαίδευσης ενώ παράλληλα παρέχει αντιπροσωπευτικό test set. Για το Memory μοντέλο, λόγω της μεγαλύτερης σταθερότητας των δεδομένων, χρησιμοποιήθηκε μικρότερο test set των 48 σημείων (1.6 ώρες), αφήνοντας περισσότερα δεδομένα για εκπαίδευση.

Τα εκπαιδευμένα μοντέλα αποθηκεύτηκαν σε pickle format για το deployment τους στο cluster, επιτρέποντας την άμεση φόρτωση και εφαρμογή τους σε production περιβάλλον χωρίς την ανάγκη επανεκπαίδευσης.

#### **3.4.3 Αξιολόγηση Μοντέλου**

##### 3.4.3.1 Απόδοση και Ερμηνεία αποτελεσμάτων

Η αξιολόγηση των δύο μοντέλων αποκάλυψε εντυπωσιακές διαφορές στην απόδοσή τους, που αντικατοπτρίζουν τη διαφορετική φύση των μετρικών που προβλέπουν. Το CPU forecasting μοντέλο επέτυχε Mean Squared Error (MSE) 0.004011 και Root Mean Squared Error (RMSE) 0.063330, με R² score 0.1293 ή 12.93%. Αν και το R² score φαίνεται χαμηλό, αυτό είναι απόλυτα κατανοητό και αναμενόμενο για τη μετρική CPU. Η χρήση επεξεργαστή σε containerized εφαrmογές χαρακτηρίζεται από υψηλή στοχαστικότητα και απρόβλεπτες αιχμές που σχετίζονται με την επεξεργασία αιτημάτων, background processes, και άλλους παράγοντες που είναι δύσκολο να μοντελοποιηθούν μόνο με ιστορικές τιμές.

Το χαμηλό RMSE του CPU μοντέλου (0.063) είναι στην πραγματικότητα πολύ ενθαρρυντικό, καθώς σημαίνει ότι παρόλο που το μοντέλο δεν μπορεί να εξηγήσει πλήρως τη διακύμανση, οι προβλέψεις του παραμένουν κοντά στις πραγματικές τιμές σε απόλυτη κλίμακα. Αυτό το καθιστά χρήσιμο για monitoring και alerting συστήματα, όπου η ανίχνευση τάσεων είναι πιο σημαντική από την απόλυτη ακρίβεια.

Εντελώς διαφορετική εικόνα παρουσιάζει το Memory forecasting μοντέλο, το οποίο επέτυχε εξαιρετικά αποτελέσματα με R² score 0.8867 ή 88.67%. Αυτό σημαίνει ότι το μοντέλο μπορεί να εξηγήσει σχεδόν το 89% της διακύμανσης στη χρήση μνήμης, κάτι που είναι εξαιρετικό για οποιαδήποτε εφαρμογή πρόβλεψης. Το MSE των 161,300,013.01 και RMSE των 12,700.39 bytes φαίνεται μεγάλο σε απόλυτη κλίμακα, αλλά όταν τεθεί σε σχέση με τη συνολική χρήση μνήμης μιας εφαρμογής (συνήθως εκατομμύρια ή δισεκατομμύρια bytes), αποδεικνύεται ότι το σφάλμα είναι σχετικά μικρό.

##### 3.4.3.2 Στρατηγική Πρόβλεψης και Practical Applications

Και τα δύο μοντέλα σχεδιάστηκαν για να προβλέπουν τα επόμενα 20 βήματα, που αντιστοιχούν σε 40 λεπτά μελλοντικών τιμών. Αυτό το χρονικό παράθυρο επιλέχθηκε στρατηγικά καθώς είναι αρκετά μεγάλο για να επιτρέπει προληπτικές ενέργειες (όπως auto-scaling ή resource allocation), αλλά όχι τόσο μεγάλο ώστε να χάνεται η ακρίβεια λόγω της αβεβαιότητας που συσσωρεύεται στις recursive προβλέψεις.

Η μέθοδος που χρησιμοποιείται είναι η recursive forecasting, όπου κάθε νέα πρόβλεψη ενσωματώνεται στα lag features για την επόμενη πρόβλεψη. Αυτή η προσέγγιση επιτρέπει τη συνεχή πρόβλεψη στο μέλλον, αλλά έχει και τον περιορισμό ότι τα σφάλματα μπορούν να συσσωρευτούν καθώς προχωράμε περισσότερο στο μέλλον.

### **3.5 Deployment των Μοντέλων**

#### **3.5.1 Δημιουργία FastAPI Services**

Δημιουργήθηκαν δύο ξεχωριστές υπηρεσίες χρησιμοποιώντας το FastAPI framework: η CPU Forecast Service που τρέχει στο port 8000 και η Memory Forecast Service στο port 8500. Και οι δύο υπηρεσίες εκθέτουν δύο κύρια endpoints που καλύπτουν διαφορετικά σενάρια χρήσης.

Το /predict endpoint σχεδιάστηκε για single-step predictions και προορίζεται για real-time scenarios όπου χρειάζεται μία γρήγορη πρόβλεψη για την επόμενη τιμή. Δέχεται ως είσοδο τα απαραίτητα lag features (15 για CPU, 8 για Memory) και επιστρέφει άμεσα την προβλεπόμενη τιμή.

Το /forecast endpoint χειρίζεται multi-step predictions χρησιμοποιώντας recursive forecasting. Επιτρέπει στον χρήστη να καθορίσει τον αριθμό των μελλοντικών βημάτων που θέλει να προβλέψει (default 20), καθιστώντας το ιδανικό για capacity planning. Η recursive φύση της πρόβλεψης σημαίνει ότι κάθε νέα πρόβλεψη χρησιμοποιείται ως είσοδος για την επόμενη.

#### 3.5.2 Ενσωμάτωση Μοντέλων

Τα εκπαιδευμένα μοντέλα (model.pkl και memory\_model.pkl) ενσωματώθηκαν απευθείας στα containers κατά το build time μέσω των Dockerfiles. Κάθε application φορτώνει το αντίστοιχο μοντέλο κατά την εκκίνηση και το κρατά στη μνήμη για γρήγορη πρόσβαση.

**3.5.3 Service Discovery και Network Configuration**

Κάθε deployment συνοδεύεται από ένα αντίστοιχο Kubernetes Service που χειρίζεται την service discovery και load balancing. Η επιλογή του NodePort service type επιτρέπει external access στις υπηρεσίες, καθιστώντας εύκολη την integration με external monitoring tools ή client applications. Αυτή η configuration είναι ιδιαίτερα χρήσιμη για testing και debugging, καθώς επιτρέπει direct access στα API endpoints από εκτός του cluster.

## 

## **4. Προκλήσεις και Περιορισμοί**

### **4.1 Kubeflow Integration**

Η αρχική πρόθεση ήταν η χρήση του Kubeflow για την ορχήστρωση του ML pipeline. Ωστόσο, αντιμετωπίσαμε πρόβλημα εγκατάστασης καθώς το Kubeflow απαιτούσε σημαντικούς πόρους οι οποίοι δεν υπήρχαν στα δικά μας μηχανήματα. Προχωρήσαμε στην χρήση Azure cloud services που μας παρείχε στην δωρεάν έκδοση 4 vCPUs, κάτι το οποίο δεν αρκούσε για την χρήση του Kubeflow καθώς αυτό ζητάει από μόνο του αυτά τα resources. Κάτι το οποίο δημιουργούσε bottleneck στο υπόλοιπο cluster.

### **4.2 Εναλλακτική Προσέγγιση**

Αντί του Kubeflow, υλοποιήσαμε το pipeline χρησιμοποιώντας:

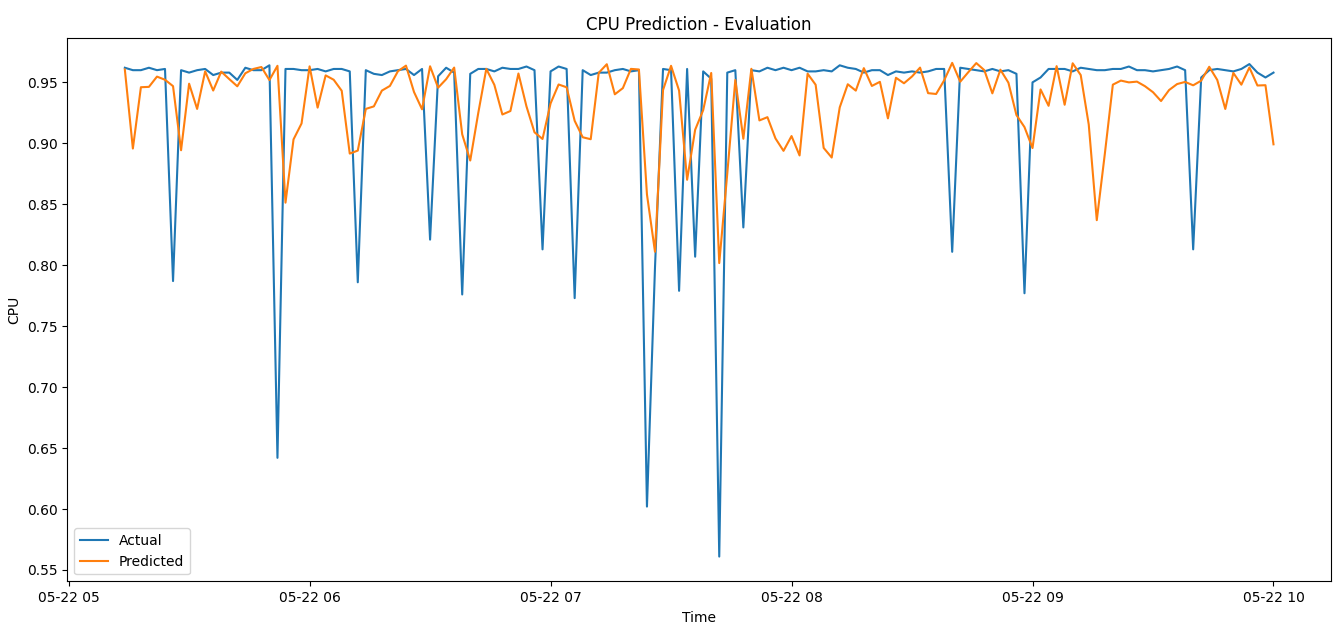
* Python scripts για την αυτοματοποίηση της ροής εργασιών
* Docker για το packaging των μοντέλων
* Kubernetes για το deployment

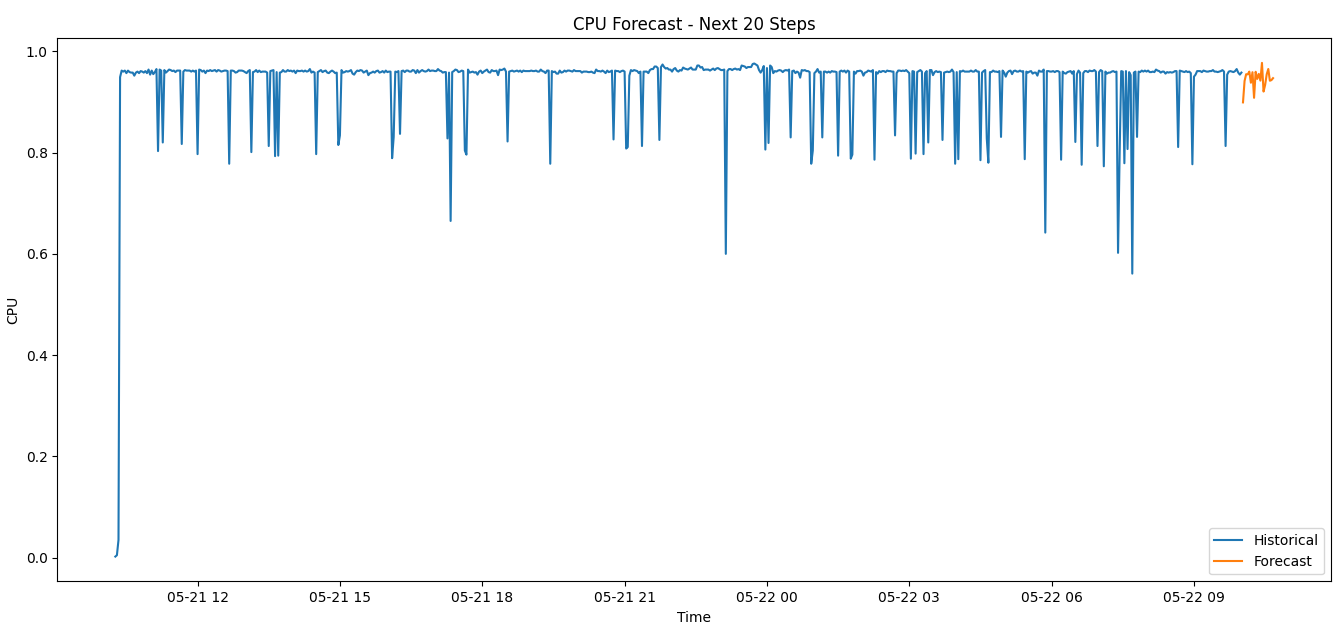
## 

## **5. Αποτελέσματα και Συμπεράσματα**

### **5.1 Επιτεύγματα**

CPU RESULTS:





MSE: 0.004011

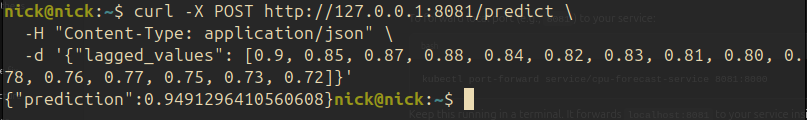
RMSE: 0.063330

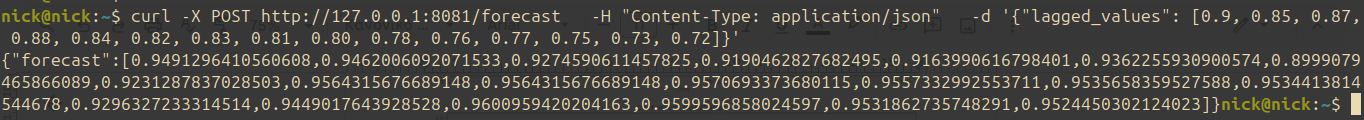
R2 Score: 0.1293

Test set size: 144 (time steps)

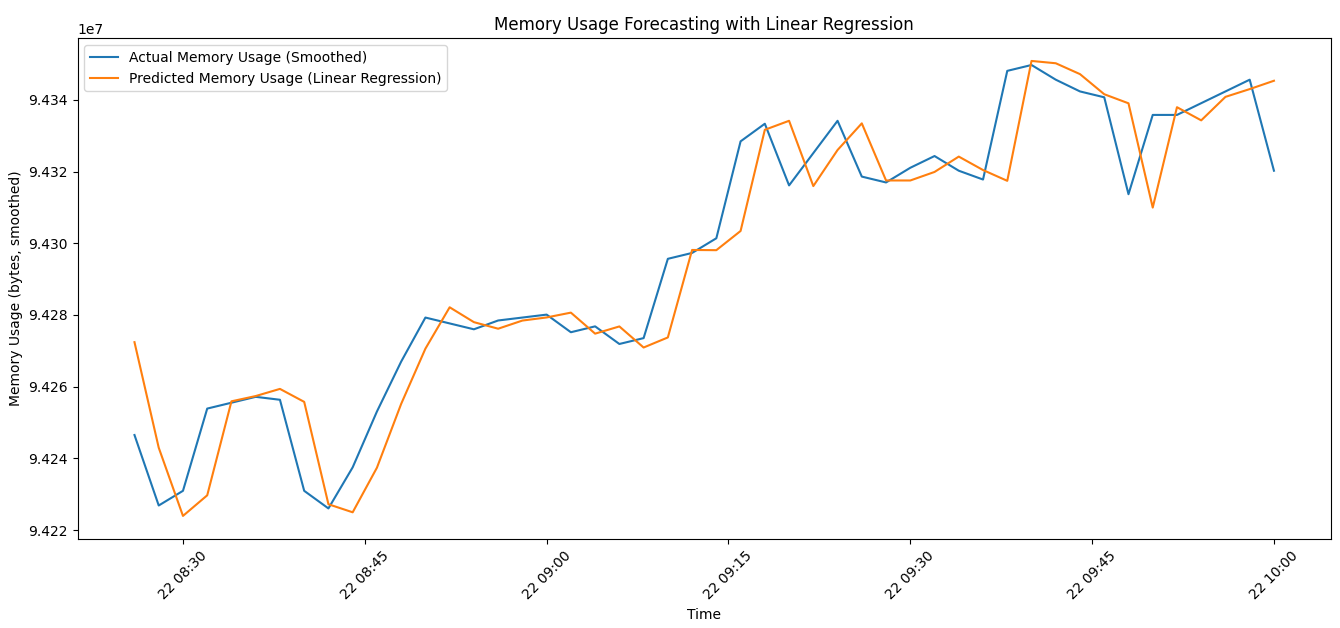
CPU SERVICE TEST:

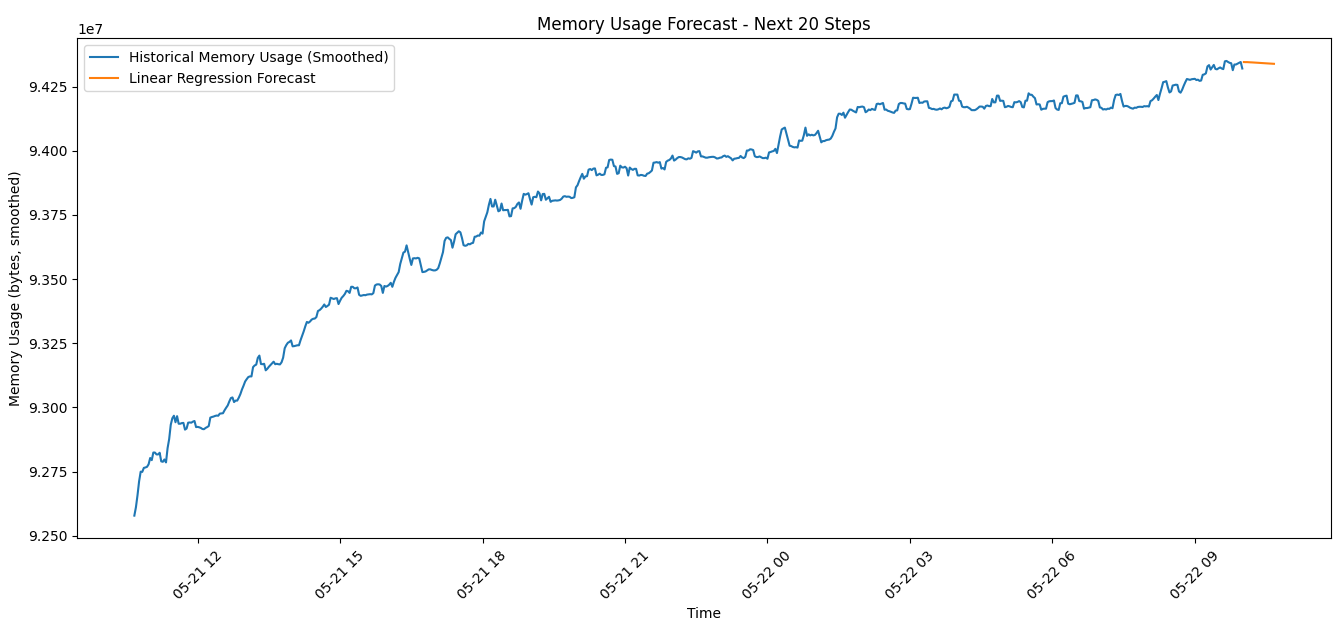
nick@nick:~$ kubectl port-forward service/cpu-forecast-service 8081:8000





MEMORY RESULTS:





MSE: 161300013.01

RMSE: 12700.39

R2 Score: 0.8867

MEMORY TEST:

nick@nick:~$ kubectl port-forward service/memory-forecast-service 8500:8500

