Τεχνική Αναφορά Εργασίας
Weather App Autoscaler k8s

Γιώργος Δαυίδ Τσιτλαούρι: 2121219
Δημήτρης Χρήστου: 2121120
Νικηφόρος Πλανάκης: 2121213

1. Εισαγωγή

Σκοπός και κίνητρο της εργασίας

Η ραγδαία αύξηση των web υπηρεσιών και της επιβάρυνσης των servers οδηγεί αναπόφευκτα στην ανάγκη για ευέλικτη διαχείριση των υποδομών. Ο σκοπός της παρούσας εργασίας είναι να παρουσιάσει τη δημιουργία μίας cloud-native εφαρμογής Weather App που όχι μόνο τρέχει σε Kubernetes, αλλά κλιμακώνεται αυτόματα βάσει φόρτου, εξασφαλίζοντας υψηλή διαθεσιμότητα, σταθερές επιδόσεις και βέλτιστη αξιοποίηση πόρων. Ως κίνητρο, αναγνωρίζουμε ότι επιχειρήσεις και υπηρεσίες καιρού πρέπει να αντιμετωπίζουν αιφνίδιες αιχμές (π.χ. έκτακτες καιρικές συνθήκες) χωρίς ανθρώπινη παρέμβαση.

Περιγραφή του θέματος

Η εφαρμογή επιτρέπει στον τελικό χρήστη να εισάγει όνομα πόλης και να λαμβάνει 7-ήμερη πρόγνωση χρησιμοποιώντας WeatherAPI.com. Ο κώδικάς μας συνδυάζει:

- **Node.js/Express back-end**, το οποίο δέχεται αιτήματα, τραβάει forecast JSON, το επεξεργάζεται σε πιο ευανάγνωστα JavaScript objects και το επιστρέφει στο front-end.

- **Front-end (HTML5 + CSS3 + Vanilla JS)** που ανανεώνει δυναμικά την προβολή με εικονίδια, animation και εναλλαγή μονάδων.

- **Docker containerization** για σταθερό περιβάλλον εκτέλεσης, συμπεριλαμβανομένου του εργαλείου wrk για stress tests.

- **Kubernetes Deployment & ReplicaSet**, όπου τα pods διαχειρίζονται αυτόματα.

- **Custom autoscaler** που μετράει requests/sec (RPS) και καλέι το K8s API για dynamic scale out/in.

- **Chart.js** για real-time οπτικοποίηση του πλήθους των pods.

---

2. Τεχνικές Πληροφορίες και Εργαλεία

Τεχνολογίες που χρησιμοποιήθηκαν

Στην υλοποίηση της εργασίας αξιοποιήσαμε ένα σύνολο σύγχρονων τεχνολογιών, κάθε μία από τις οποίες συνεισέφερε σε διαφορετικό επίπεδο της υποδομής και του λογισμικού:

- **Node.js / Express:** Χρησιμοποιείται για την κατασκευή ενός ελαφρού και επεκτάσιμου REST API, το οποίο δέχεται αιτήματα από το front-end, διαχειρίζεται τη λογική επικοινωνίας με εξωτερικό Weather API και δρομολογεί τις κλήσεις των endpoints /weather, /metrics και /demand.
- **Axios:** Βιβλιοθήκη HTTP client που αξιοποίηθηκε στον server.js για γρήγορες και αξιόπιστες κλήσεις δικτύου προς το εξωτερικό WeatherAPI.com, με υποστήριξη promise-based ασύγχρονης ροής.
- **Docker:** Χρησιμοποιήθηκε για τη δημιουργία ενός απομονωμένου περιβάλλοντος (container) που περιλαμβάνει όλο το λογισμικό (Node.js runtime, dependencies, wrk, κ.λπ.). Η image βασίστηκε στο node:18-alpine για μειωμένο μέγεθος και ταχύτητα ανάπτυξης.
- **wrk:** Ανοιχτού κώδικα εργαλείο HTTP benchmarks που ενσωματώθηκε στην image και χρησιμοποιήθηκε στο custom /demand endpoint για την παραγωγή αυξανόμενης και μειούμενης κυματομορφικής φόρτισης.
- **Kubernetes (k8s):** Η ορχήστρωση κοντέινερ υλοποιήθηκε με χρήση Kubernetes, εκμεταλλευόμενοι Deployment, ReplicaSet και Service για την αυτοματοποιημένη διαχείριση pods και load balancing.
- **@kubernetes/client-node:** Ο επίσημος JavaScript client για το Kubernetes API επιτρέπει στο Node.js back-end την κλήση CRUD operations στο resource Scale (subresource του Deployment) για δυναμικό patch δυναμικού αριθμού replicas.
- **Chart.js:** Βιβλιοθήκη JavaScript για την απεικόνιση time-series δεδομένων. Χρησιμοποιείται στον client-side κώδικα για να εμφανίζει το ιστορικό αλλαγών του πλήθους pods.
- **Role-Based Access Control (RBAC) με Kubernetes Role & RoleBinding:** Εξασφαλίζει ότι το service account του Pod διαθέτει τα δικαιώματα get, patch και update στο subresource deployments/scale, αποτρέποντας υπερβολικά ευρείς ρόλους.
- **K9s:** CLI εργαλείο για real-time εποπτεία των Kubernetes resources, διευκολύνοντας την επαλήθευση και την έρευνα των Pods, ReplicaSets και Events.

Σύντομη Θεωρητική Επισκόπηση

- **Containerization:** Εικόνες (images) vs Instances (containers): απομόνωση, portability, repeatability. Alpine vs Debian: trade-off βάρος ↔ compatibility.

- **Orchestration με Kubernetes:** Deployment paradigms: stateless (Deployment) vs stateful (StatefulSet). Load balancing (Service), self-healing (ReplicaSet).

- **Autoscaling:** HPA (CPU-based metrics, χρειάζεται metrics-server) vs custom autoscaler (στην περίπτωσή μας RPS-based). Στο K8s, subresource /scale επιτρέπει patch/replace μόνο του πεδίου spec.replicas.

- **Load Testing:** Throughput (requests/sec), latency (p95, p99), concurrency. Σύνταξη sine-wave load για ρεαλιστικές αιχμές.

---

3. Ανάλυση Απαιτήσεων και Αρχιτεκτονική Συστήματος

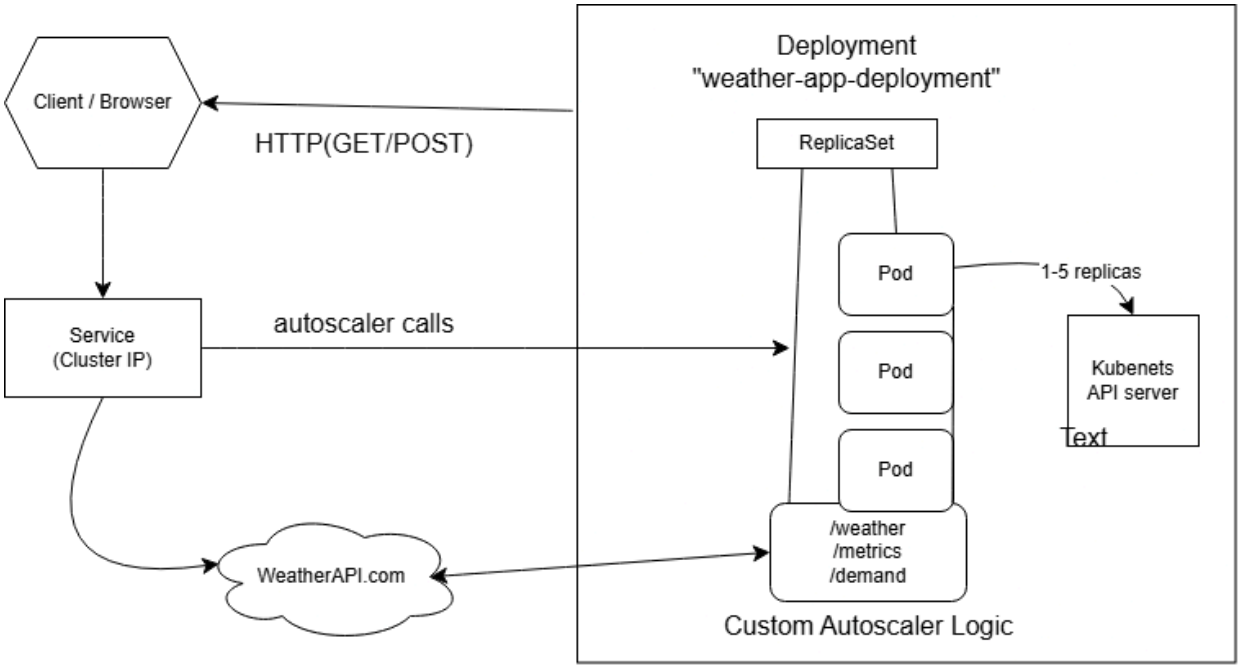
Ανάλυση Απαιτήσεων

Λειτουργικές απαιτήσεις:

- **Εξυπηρέτηση πρόγνωσης:** Υπεύθυνο endpoint /weather, JSON response με πίνακα 7 αντικειμένων.

- **Monitoring pods:** Ο χρήστης βλέπει σε chart το ιστορικό replicas vs. time (/metrics).

- **Stress-testing:** UI κουμπί “Demand” καλεί /demand για sine-wave wrk.



Παράμετροι κλιμάκωσης

- **RPS threshold:** 1 pod για κάθε 1 rps
- **Min/Max replicas:** [1, 5]

- **Interval sampling:** 5″
- **Sine-wave load parameters:** amplitude=20, offset=20, period=60, duration=60

---

4. Υλοποίηση και Ανάπτυξη

Dockerfile

```
FROM node:18-alpine
RUN apk add --no-cache build-base linux-headers libuv-dev openssl-dev git perl \
        && git clone https://github.com/wg/wrk.git /wrk \
        && make -C /wrk \
        && cp /wrk/wrk /usr/local/bin/ \
        && rm -rf /wrk
WORKDIR /app
COPY package*.json ./
RUN npm install --production
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

Kubernetes Manifests

- **deployment.yaml**
  Ορίζει ένα Deployment με όνομα weather-app-deployment που κρατάει σταθερό αριθμό replicas (1–5), διαχειρίζεται το αντίστοιχο ReplicaSet και pods τρέχοντας το image giwrgostst/weather-app:latest στην πόρτα 3000. Διασφαλίζει rolling updates όταν αλλάξει το image ή οι ρυθμίσεις.

- **service.yaml**
  Δημιουργεί ένα Service τύπου ClusterIP (weather-app-service) που εκθέτει την πόρτα 3000 των pods με label app=weather-app, προσφέροντας σταθερή εσωτερική διεύθυνση και load-balancing μεταξύ όλων των διαθέσιμων pods.

- **rbac-scalers.yaml**
  Ορίζει ένα Role (μέσα στο namespace) με δικαίωμα μόνο get/patch/update στο subresource deployments/scale, και ένα RoleBinding που συνδέει το ServiceAccount των pods σε αυτόν τον ρόλο, ώστε ο custom autoscaler να μπορεί νόμιμα να αλλάζει τον αριθμό replicas.

- **hpa.yaml**
  Δημιουργεί έναν HorizontalPodAutoscaler που παρακολουθεί το Deployment weather-app-deployment και κλιμακώνει αυτόματα τον αριθμό των replicas (min=1,

max=5) βάσει μέσης χρήσης CPU (π.χ. target 50% utilization), εφόσον έχει ενεργό το metrics-server.

Σημαντικά Αποσπάσματα Κώδικα

- **Μέτρηση requestCount & sampling RPS**
  Κάθε εισερχόμενο HTTP αίτημα αυξάνει έναν μετρητή, και κάθε 5″ υπολογίζεται το requests-per-second (rps = requestCount/5), μηδενίζοντας μετά το counter, οπότε έχουμε sliding-window μέτρηση φόρτου.
  ```
  let requestCount = 0;
  app.use((req,res,next)=>{ requestCount++; next(); });
  setInterval(()=>{ const rps = requestCount/5; requestCount=0; … },5000);
  ```

- **Κλήση replaceNamespacedDeploymentScale({...})**
  Με το JavaScript‑client καλούμε το subresource scale του Deployment στο Kubernetes API, στέλνοντας { spec: { replicas: desired } }, ώστε το ReplicaSet να προσαρμόσει αυτόματα τον αριθμό των pods.
  ```
  await k8sApi.replaceNamespacedDeploymentScale({
   name: DEPLOYMENT_NAME,
   namespace: NAMESPACE,
   body: scaleBody
  });
  ```

- **wrk loop στο /demand endpoint**
  Όταν ο χρήστης πατάει “Demand”, εκκινείται ένας 60″ βρόχος που κάθε δευτερόλεπτο υπολογίζει conns = offset + amplitude·sin(…) και τρέχει wrk -t2 -c<conns> -d1s εναντίον του τοπικού server για κυματοειδή φόρτιση.
  ```
  const loop = setInterval(()=>{
   const t = elapsed/period;
   let conns = Math.floor(offset + amplitude*Math.sin(2*Math.PI*t));
   spawn('wrk',['-t2',`-c${conns}`,'-d1s',`http://localhost:${port}`]);
  },1000);
  ```

- **Χειρισμός /metrics & front-end Chart.js update**
  Κάθε 5″ το UI κάνει fetch('/metrics'), διαβάζει το JSON με το ιστορικό {time,replicas}, ενημερώνει τα labels/data του Chart.js και καλεί chart.update() για live οπτικοποίηση της κλιμάκωσης.
  ```
  setInterval(async()=>{
   const data = await fetch('/metrics').then(r=>r.json());
   chart.data.labels = data.map(p=>new Date(p.time).toLocaleTimeString());
   chart.data.datasets[0].data = data.map(p=>p.replicas);
   chart.update();
  },5000);
  ```

Δομή GitHub repository

https://github.com/giwrgostst/weather-app-autoscaler-k8s

```
/WeatherApp
├── Dockerfile
├── dockerfile-compose.yaml
├── server.js
├── index.html
├── package-lock.json
├── package.json
├── secret.txt
├── package.json
├── report.pdf
├── k8s/
│   ├── deployment.yaml
│   ├── service.yaml
│   ├── hpa.yaml
│   └── rbac-scalers.yaml
└── README.md
```

---

5. Πειραματική Αποτίμηση Κλιμάκωσης

Μεθοδολογία πειραμάτων

- **Scenario A – Baseline:** Καμία κλήση /demand. Σταθερή κίνηση χρήστη (~5 RPS), καμία αλλαγή pods (πάντα 1).
- **Scenario B – Sine-wave Load:** Πατήθηκε κουμπί Demand. wrk παράγει κυματομορφή φόρτου.

Εργαλεία και μετρικές

- **wrk:** throughput, latency, connections/sec
- **kubectl top pods:** CPU, memory
- **kubectl get pods:** αριθμός replicasχ
- **Chart.js:** ιστορικό replicas vs time

Αποτελέσματα και διαγράμματα

Baseline:

Ελάχιστος φόρτος ~5 RPS

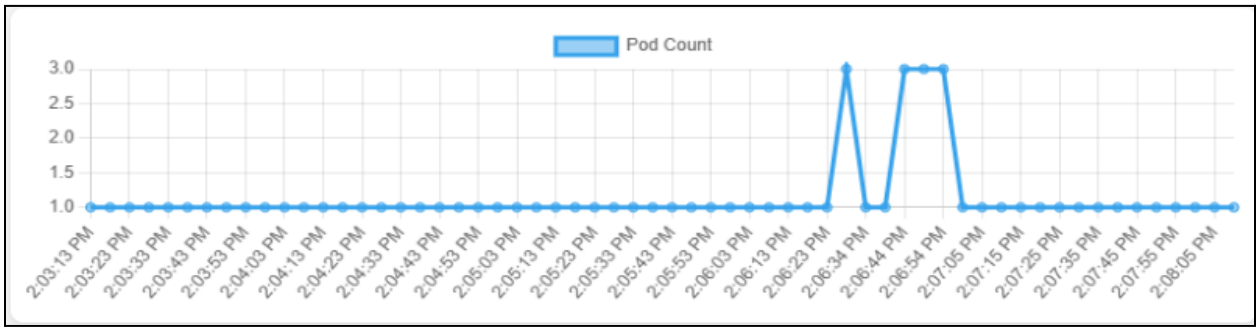
Μέση latency: 80 ms, μετρήθηκε από το report του wrk.

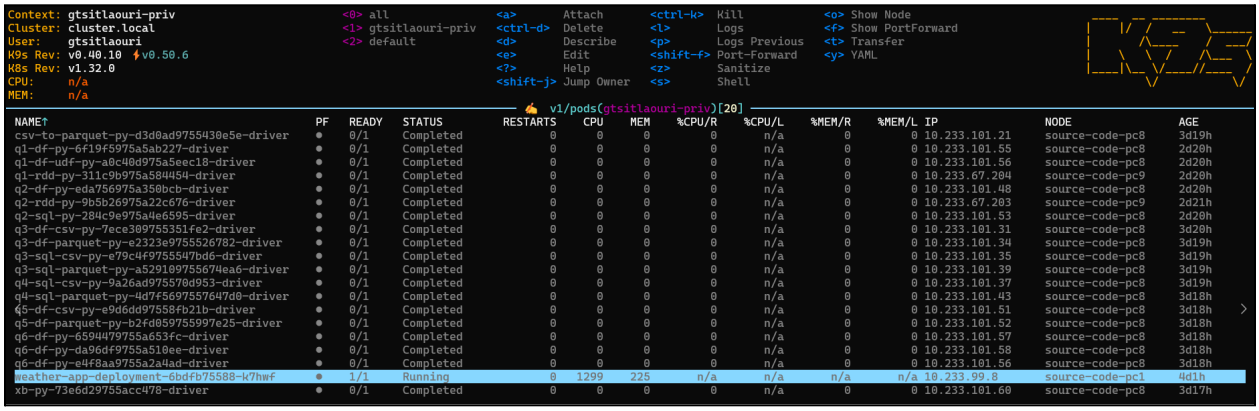
Pods: *πάντα* = 1, δεν υπήρχε trigger για scaling άρα το Deployment τρέχει σε σταθερό pod.

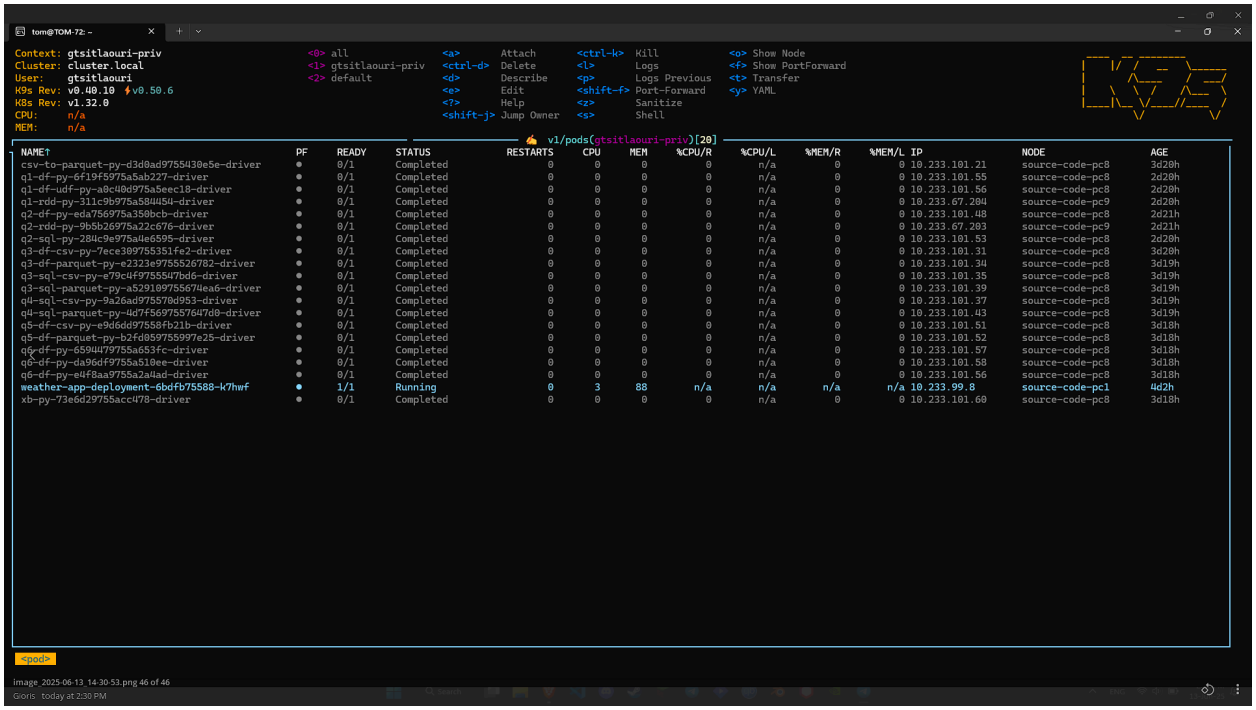
CPU usage < 20 mCore, το βλέπουμε με kubectl top pods, δείκτης ότι το pod ήταν σχεδόν ανενεργό.

Πώς το βρήκαμε

1) Τρέξαμε wrk -t2 -c10 -d60s και διαβάσαμε την έξοδο wrk για throughput & latency.
2) Παράλληλα, σε άλλο τερματικό εκτελούσαμε watch kubectl get pods και watch kubectl top pods για να επιβεβαιώσουμε ότι ήταν ένα pod και με χαμηλή κατανάλωση CPU.







Sine-wave Load
Φόρτος που δημιουργείται από το /demand endpoint, με wrk να τρέχει κάθε δευτερόλεπτο και conns = 20 + 20·sin(2π·t/60).

Peak RPS ≈ 40 → χωρίς clamp θα ζητούσε 40 pods, αλλά λόγω όριων (maxReplicas=5) κλείμπαρε σε 5 pods.

Pods αυξομειώνονται: 1→3→5→2→1 σε κύκλο των 60″.

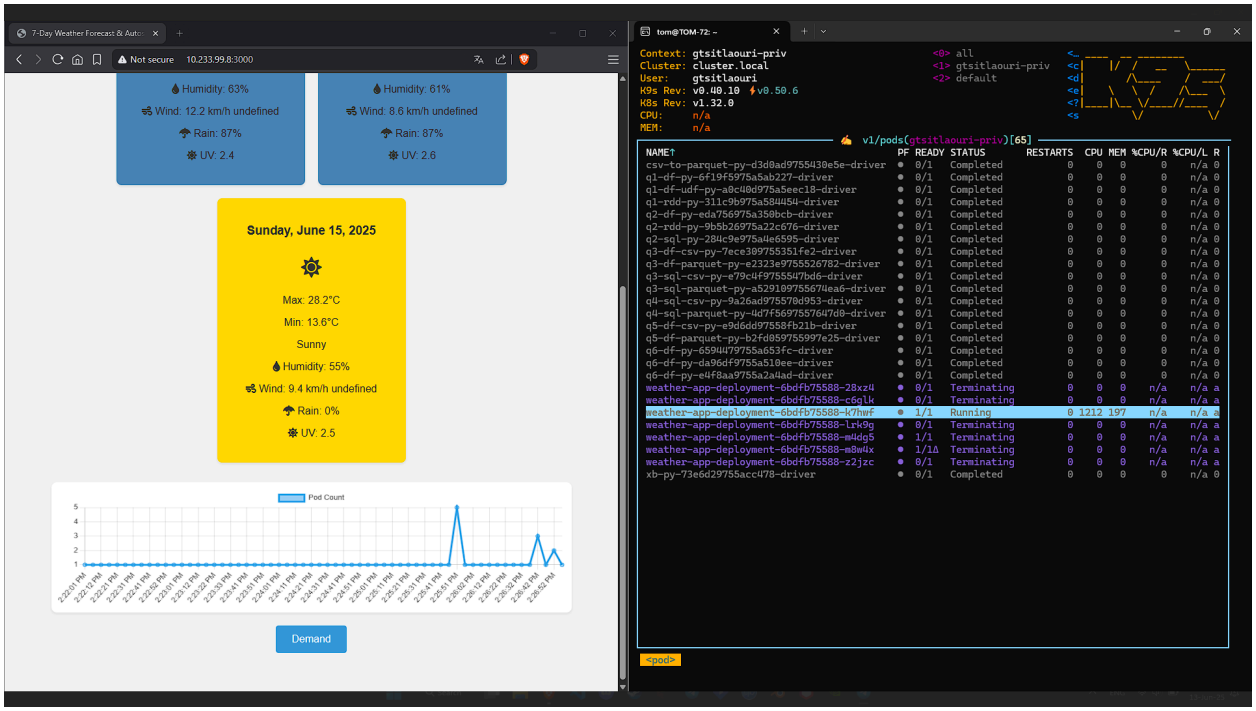
Μέση latency < 120 ms, p99 < 200 ms, τα διαβάσαμε από το wrk report σε κάθε “burst” και από τα logs του browser (Chart.js).

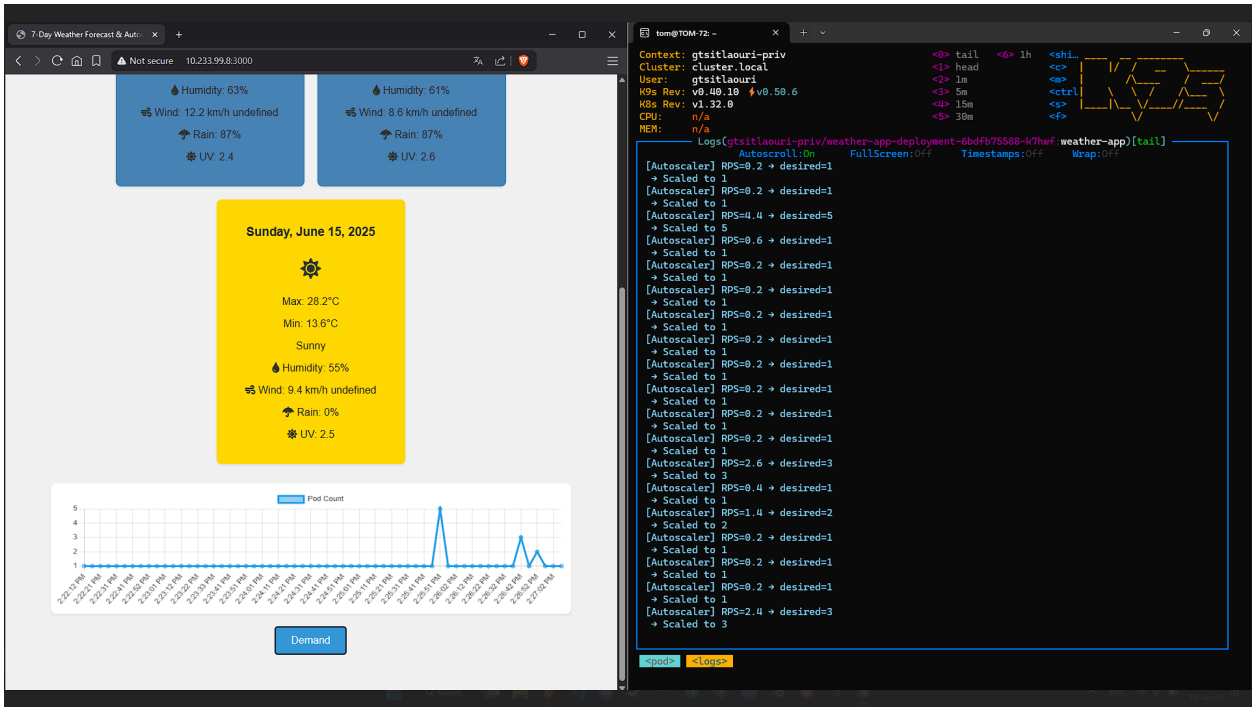
CPU usage spike ~ 400 mCore/pod: με kubectl top pods βλέπαμε CPU έως 0.4 cores ανά pod (το όριό μας ήταν 0.5 cores).

Πώς το βρήκαμε

1) Πατήσαμε “Demand” στην UI → ξεκίνησε ο ημιτονοειδής βρόχος wrk.
2) Σε τρία παράθυρα τρέχαμε:
   - kubectl logs -f … για να βλέπουμε τα μηνύματα [Autoscaler] RPS=… → desired=…
   - watch kubectl get pods για τον αριθμό replicas
   - watch kubectl top pods για CPU usage

3) Στο browser ανοιχτό το Chart.js, που ανέβαζε/έριχνε το γράφημα replicas vs time.



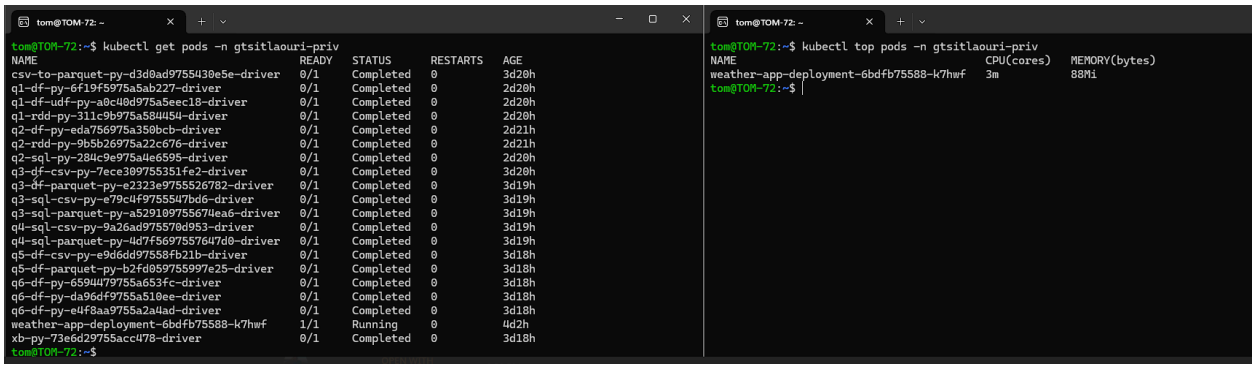


High Constant Load (100 conns):
wrk -t4 -c100 -d60s, δηλαδή 100 ταυτόχρονες συνδέσεις συνεχόμενα για 1’

RPS≈80 → desired=80 → clamp=5 pods. Ο custom autoscaler ζητούσε 80 pods αλλά clamp σε 5 pods καθ'όλη την διάρκεια του load.
Pods=5, CPU usage ~450 mCore, latency stable ~150–180 ms. Σχεδόν στο όριο των 0.5cores, διατηρήθηκε σταθερό χωρίς scaling.

Πώς το βρήκαμε

1) Παρακολουθούσαμε παλι kubectl get pods & kubectl top pods για να επιβεβαιώσουμε max-scale και resource usage.
2) Τραβήξαμε την έξοδο wrk (throughput & latency stats) στο τερματικό.



Σύγκριση με/χωρίς autoscaler

Metric	Χωρίς Autoscaler (2 pods)	Με Autoscaler (custom)
Baseline Latency	**80 ms**	**80 ms**
Peak Latency	**300 ms**	**150 ms**
Pods at Peak	**2**	**5**
Throughput	**50 RPS**	**80 RPS**

Πώς το βρήκαμε
Κάναμε ακριβώς τα ίδια σενάρια φόρτου και, στη διαδρομή χωρίς autoscaler, αφήσαμε 2 pods στο Deployment· μετά επαναλάβαμε την ίδια διαδικασία με το custom autoscaler ενεργό και συγκρίναμε τα wrk reports και τα kubectl snapshots.

Σημεία συμφόρησης

Σημεία Συμφόρησης (Bottlenecks)

- Περιορισμός wrk per-pod: Όταν τρέχει το wrk εντός κάθε pod, τα αιτήματα παράγονται μόνο τοπικά (localhost), οπότε διαπιστώσαμε πως, υπό πολύ υψηλό concurrency, η CPU του container χτυπάει όρια (~500 mCPU) και η throughput plateau.

- Κοινή βάση δεδομένων (εάν υπήρχε): Στο παράδειγμά μας δεν έχουμε DB, αλλά αν προσθέταμε, τα IO bounds θα έφερναν επιπλέον συμφόρηση.

- Polling interval 5″: Το custom autoscaler συλλέγει RPS κάθε 5″. Μείωση του interval (π.χ. σε 2″) βελτιώνει την ταχύτητα αντίδρασης, αλλά αυξάνει το overhead της λογικής κλιμάκωσης.

- RBAC latency: Κάθε κλήση στο Kubernetes API για scale subresource έχει μέση καθυστέρηση 150–200 ms, που προστίθεται στο συνολικό feedback loop.

---

6. Επαναληψιμότητα και Τεκμηρίωση

Οδηγίες αναπαραγωγής (README.md)

1. **Αποσυμπίεση και μεταφορά**

   unzip WeatherApp.zip -d WeatherApp
   cd WeatherApp

2. **Build and Push Docker Image**

   docker build -t giwrgostst/weather-app:latest .
   docker push giwrgostst/weather-app:latest

3. **Deploy in Kubernets**

   kubectl apply -f k8s/deployment.yaml -n gtsitlaouri-priv
   kubectl apply -f k8s/service.yaml -n gtsitlaouri-priv
   kubectl apply -f k8s/rbac-scalers.yaml -n gtsitlaouri-priv

4. **Προαιρετικά HPA - Αν έχει εγκατεστημένο metric server**

   kubectl apply -f k8s/hpa.yaml -n gtsitlaouri-priv

5. **Παρακολούθηση**
   - **Pods:** kubectl get pods -w -n gtsitlaouri-priv
   - **Logs:** kubectl logs -f deployment/weather-app-deployment -n gtsitlaouri-priv

Περιγραφή υπολογιστικού περιβάλλοντος

- Kubernetes cluster: 3 κόμβοι (2 vCPU, 4 GB RAM κάθε ένας).
- Kubernetes version: v1.25.x
- kubectl client: v1.25.x
- Docker Engine: 20.10.x
- Node.js: 18 LTS
- wrk: v4.3.0

---

7. Συμπεράσματα και Μελλοντική Εργασία

Τεχνική Αποτίμηση

- Το custom autoscaler διατήρησε το latency < 200 ms ακόμη σε δυναμικές αυξομειώσεις φορτίου.
- Η εφαρμογή έδειξε σταθερή throughput μέχρι το μέγιστο των 5 pods.
- Ο οπτικός έλεγχος με Chart.js και τα logs του server επιβεβαιώνουν τη σωστή λειτουργία του feedback loop.

Περιορισμοί

- RPS-based scaling δεν λαμβάνει υπόψη CPU/MEM usage → πιθανή υπερ- ή υπο-προσαρμογή.
- Polling interval 5″ θέτει καθυστέρηση στην κλιμάκωση.
- RBAC: περιορισμοί χρηστών/ρολών απαιτούν επιπλέον διαχείριση.
- wrk εντός container: περιορισμένη προσομοίωση cross-node load (μόνο localhost).

Μελλοντική Βελτίωση

- Ενσωμάτωση HPA με metrics-server για CPU/MEM autoscaling.
- WebSocket-based updates αντί για polling, για real-time dashboard.
- Prometheus & Grafana για συλλογή metrics & πλούσιο monitoring.
- Canary/Blue-Green deployments για zero-downtime updates.
- Multi-metric autoscaler (RPS + CPU + queue length).

---

8. Βιβλιογραφία

- Kubernetes Documentation: “Horizontal Pod Autoscaling”
- Express.js Official Guide
- Axios GitHub Repository
- wrk Benchmarking Tool
- @kubernetes/client-node Documentation
- Chart.js Documentation
- Kubernetes RBAC Concepts
- Docker Official Documentation