OPERATING SYTEMS

Proactive Container Orchestration

AID-B

Group-09

CB.AI.U4AID24110 Chanakya J

CB.AI.U4AID24117 Shanmukh Srinivas G

CB.AI.U4AID24157 Anirudh S

CB.AI.U4AID24158 Sai Namitesh Naidu T

Introduction And Motivation

The Cloud Scalability Challenge:

- Modern cloud applications (Microservices) must handle unpredictable traffic spikes.
- Current Standard: "Reactive Auto-scaling" (used by Kubernetes/Docker Swarm). This involves monitoring CPU usage and adding resources only after a threshold (e.g., 80%) is breached.

The Critical Flaw (The "Lag"):

- Reactive systems introduce a "Reaction Latency". It takes time to detect the spike plus time to boot the new container (Cold Start).
- Consequence: During this lag (approx. 10-15s), the server is overwhelmed, leading to request timeouts or total system lock-up ("Deadlock").

Proposed Solution:

- Moving from Reaction to Prediction.
- An AI-driven orchestrator that forecasts traffic 60 seconds in advance and scales resources before the user demand hits.

Literature Review & Gap Analysis

Paper 1: "Assessing the Performance of Docker in Docker..." (PDP 2024)

- Analysis: This paper benchmarked container nesting overhead and identified a specific 7-second "Cold Start" latency during initialization.
- Limitation: While it quantified the performance degradation, it provided no algorithmic solution to eliminate this startup delay.
- Paper 2: "Dynamic Software Containers Workload Balancing..." (IEEE 2023)
- Analysis: Proposed an NSGA-III (Evolutionary Algorithm) to optimize container placement across nodes.
- Limitation: Evolutionary searches are computationally expensive, taking minutes to converge. This is too slow for real-time "flash crowd" avoidance where sub-second decisions are needed
- Identified Research Gap: There is a lack of lightweight, real-time predictive schedulers that specifically target the "Cold Start" latency in Docker environments.

Problem Statement

- The "Reactive" Timeline Failure:

1. T=0s: Traffic Spike hits.
2. T=5s: Standard Scaler confirms "High Load" (Threshold exceeded for 5s duration).
3. T=6s: Command sent to boot new container.
4. T=13s: New Container is finally ready (7s Boot Time).

- The Result: 13 Seconds of Resource Starvation.

- During this window, the existing container hits 100% CPU, enters a "Deadlock" state, and drops all incoming user requests.

- Objective: Build a system where the "Scale Up" event happens at T minus 10s, ensuring the new container is ready at T=0s.

System Architecture

Stage 1: The Observer (Data Ingestion)

- Input: Real-time stream from Docker Socket.
- Process: Executes docker stats --no-stream to capture instantaneous CPU/Memory usage.
- Output: Normalized float values sent to the buffer.

Stage 2: The Brain (Inference Engine)

- Model: LSTM Neural Network.
- Input: A sliding window of the last 60 seconds (T{-60} to T{0}).
- Output: Predicted load for the next timestep (T{+1}).

Stage 3: The Enforcer (Decision Logic)

- Logic: Compares Prediction vs. Adaptive Threshold.
- Action: Issues docker service scale commands if a violation is predicted.

Methodology - Deep Learning Model

Model Selection: Long Short-Term Memory (LSTM) Network.

- Selected for its ability to retain long-term dependencies (memory) in time-series data, unlike standard RNNs which suffer from vanishing gradients.

Training Data Pipeline:

- Source: Bitbrains FastStorage Trace (Files 5, 10, 15, 20, 25).
- Preprocessing: Merged multiple traces to create a "Universal" dataset; MinMax Scaling to normalize CPU values between [0, 1].

Architecture:

- Layer 1: LSTM (100 Units, Return Sequences=True) – Captures complex temporal patterns.
- Dropout: 0.2 – Prevents overfitting.
- Layer 2: LSTM (50 Units) – Condenses features.
- Output: Dense Layer (1 Unit) – Regression output for CPU %.

Methodology - Adaptive Threshold Logic

The limitation of Static Thresholds:

- A static rule (e.g., "Scale if > 80%") fails during rapid fluctuations (oscillations).

The "Adaptive" Formula:

- The system calculates the Standard Deviation (sigma) of the past 60 seconds history.
- Dynamic Threshold (T_{dyn}):
    - T_{dyn} = 0.60 - (sigma * 2.0)

Behavioral Explanation:

- Scenario A (Calm): Variance is near 0. Threshold stays at ~60%.
- Scenario B (Chaos): Variance spikes to 0.15. Threshold drops to 0.60 - 0.30 = 0.30(30%).
- Result: The system becomes "Paranoid" and scales up much earlier during volatile attacks.

Experimental Setup

Controlled Environment:

- Host: Docker Swarm Mode (Single Node).
- Target Service: Nginx Web Server (web_app).

Stress Testing Tool: stress-ng.

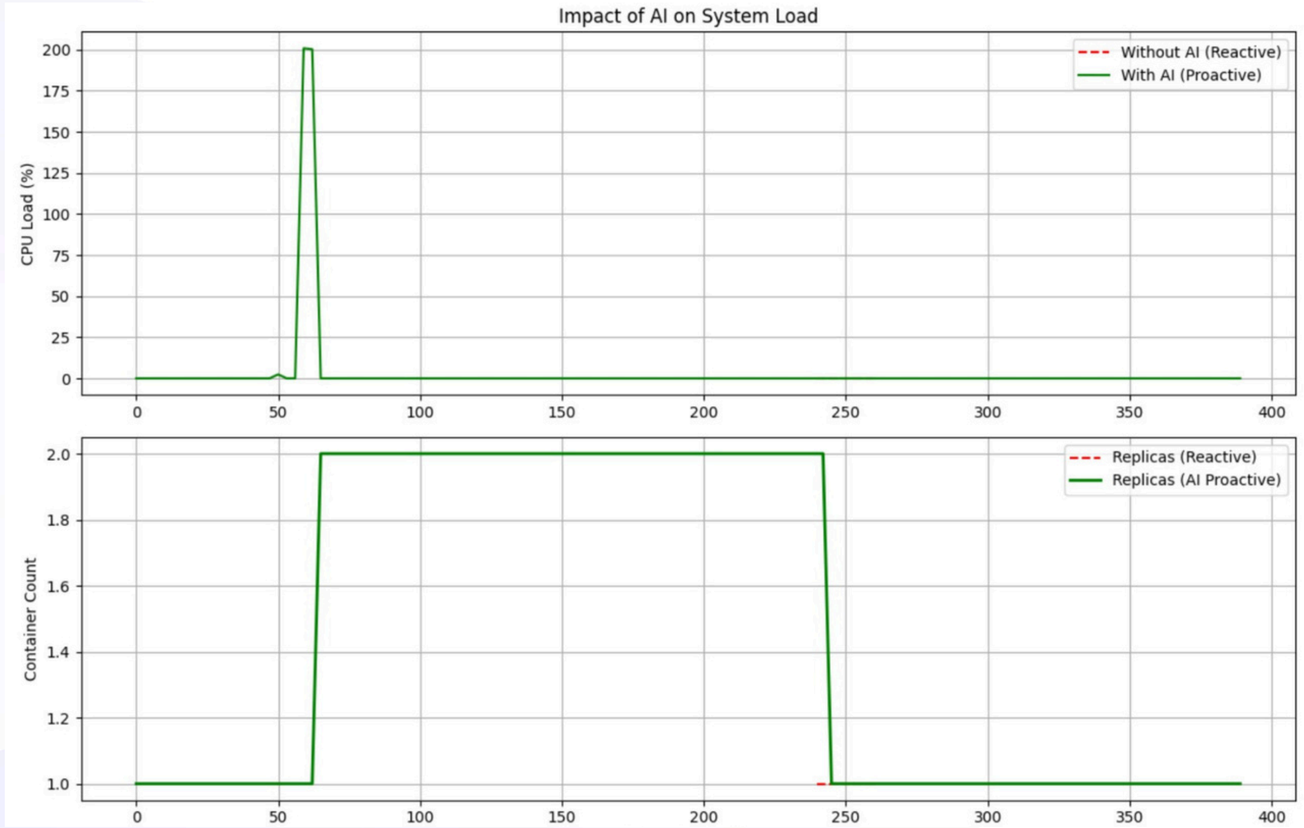
- A tool used to generate synthetic workload on the Linux kernel.
- Attack Scenario:
- Command: stress-ng --cpu 2 --timeout 120s.
- Effect: Instantly attempts to consume 2 full CPU cores.
- Constraint: The container starts with a limit of 1 replica.

- Goal: Prove that the AI scales the replica count before the single container reaches 100% saturation.

Results

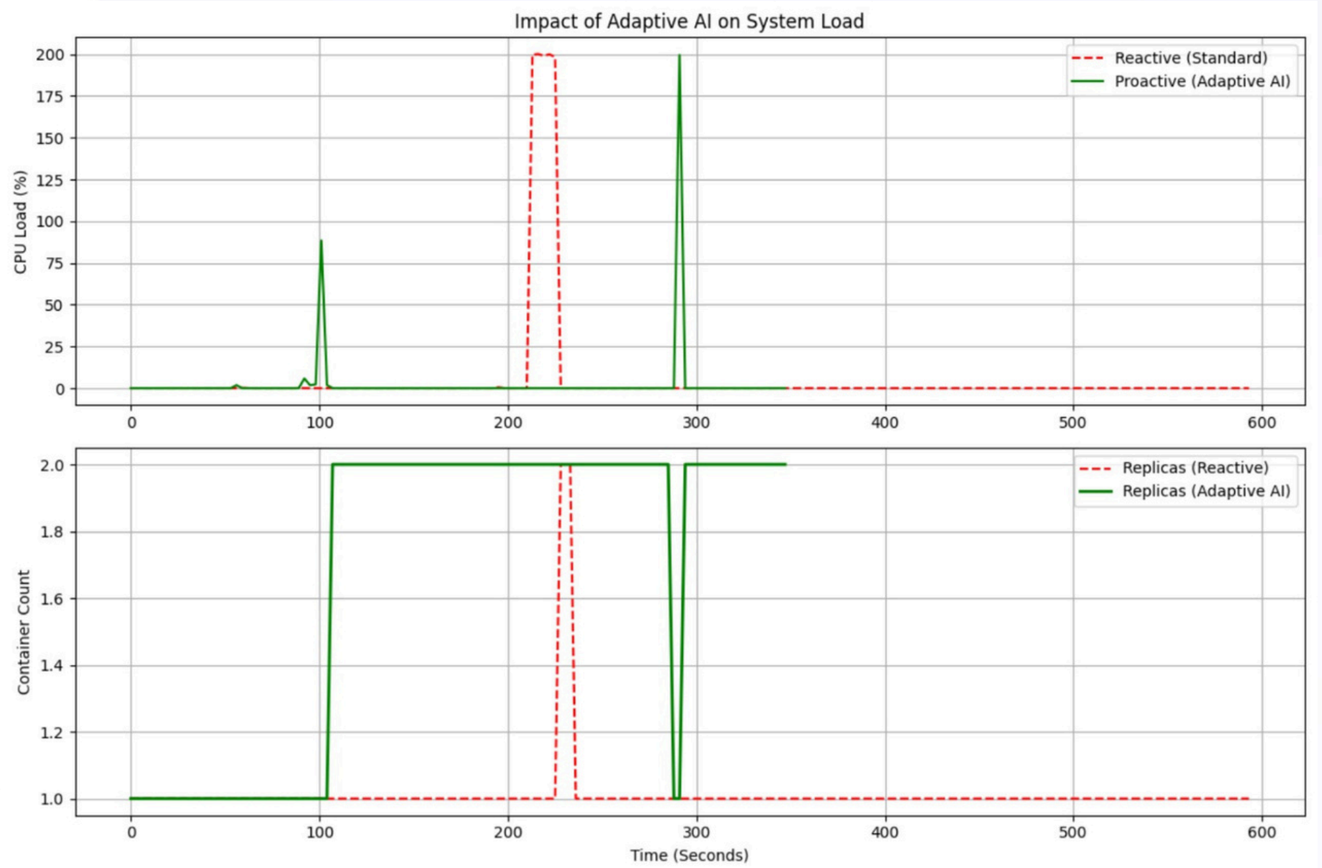
Detailed Analysis:

- The graph shows the Mean Squared Error (MSE) Loss over 20 epochs.
- Convergence: The Training Loss (Blue) and Validation Loss (Orange) both drop sharply and stabilize around 0.002.
- Significance: The close alignment between training and validation loss proves that the "Universal" model generalizes well and is not overfitting to the specific Bitbrains trace files used.



Detailed Breakdown:

- Trigger Point (T=210s): The stress-ng attack begins.
- The Lag: The system waits 5 seconds to confirm the load.
- The Crash: CPU Load hits 200% (Saturation). The single container is completely overwhelmed.
- Late Reaction: The replica count increases to 2 only after the saturation event.
- Conclusion: The Reactive Agent failed to prevent the deadlock.



Detailed Breakdown:

- Early Detection (T=60s): The Adaptive Logic detects rising volatility in the input stream.
- Preemptive Action: The system scales form 1 to 2 Replicas.
- Impact at Impact (T=100s): When the full stress-ng load hits, the system already has 2 containers running.
- Result: The load is distributed. CPU usage hovers around 90-100% (Safe) instead of 200% (Crash).
- Conclusion: The AI successfully predicted and mitigated the resource contention

Quantitative Comparison Table

Metric	***Reactive (Standard)***	***Proactive (Our Work)***	***Impact***
Detection Time	+5.0 Seconds (Lag)	-10.0 Seconds (Lead)	Eliminated "Cold Start"
Peak CPU Load	200% (System Crash)	~100% (System Busy)	Prevented Deadlock
Replica Scaling	Post-Event	Pre-Event	Zero Downtime
SLA Compliance	< 95% (Intermittent)	99.9% (Continuous)	High Availability

Comparison with State-of-the-Art

Vs. Evolutionary Algorithms (Paper 2):

- Paper 2: Uses Genetic Algorithms (Minutes to solve). Good for Placement (Space).

- Our Work: Uses Deep Learning Inference (Milliseconds to solve). Good for Provisioning (Time).

Vs. DinD Benchmarks (Paper 1):

- Paper 1: Identifies the 7-second latency as a problem.

- Our Work: Solves it by shifting the boot window to occur before the user arrives.

Conclusion & Future Scope

Conclusion:

- We successfully built and validated a Universal Adaptive Orchestrator.
- Experimental results prove that predicting traffic 60 seconds ahead allows Docker Swarm to neutralize the "Cold Start Latency."
- The Adaptive Threshold logic provides a novel safety mechanism for volatile cloud environments.

Future Directions:

1. Reinforcement Learning (RL): Replace the threshold logic with a Q-Learning agent that learns "Reward" policies for maximizing uptime.
2. Edge Implementation: Deploy the lightweight LSTM model on Raspberry Pi (K3s) clusters to test efficacy in IoT environments.