International Journal of AI, BigData, Computational and Management Studies
Noble Scholar Research Group | Volume 5, Issue 1, PP 145-158, 2024
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I1P115

ML Algorithms that Dynamically Allocate CPU, Memory, and I/O Resources

Nagireddy Karri
Senior IT Administrator Database, Sherwin-Williams, USA.

Abstract: The cloud and edge platforms of the present time need to distribute CPU, memory, and I/O under unsteady, multi-tenanted workloads without violating service-level aims (SLOs), finances, and energy restrictions. This paper introduces a safety-first architecture that integrates the telemetry-based prediction and policy-based decision making. We have instrumented hosts and services with fine-grained signals CPU runnable depth and throttling, page-fault rate and cache residency, disk/NVMe queue depth and tail latency, NIC retransmits and engineer cross-resource features to measure the coupling effects. Developing these contributions, we consider three categories of the methods: (i) the supervised forecasters (gradient boosting, temporal CNN/Transformer) to predict demand in the short-horizon and SLO-risk, (ii) the constrained reinforcement learning to predict joint and continuous changes in the shares/quota of CPU, memory constraints/placement, and I/O priorities, (iii) the hybrid predict-then-optimize controllers with model-predictive guardrails. Automatic rollback and limited step sizes of canary-first rollout guarantee stability of production. Our solution to trace-driven experiments on OLTP microservices, streaming analytics, storage-heavy phases, and ML inference pipelines results in 20-45% p99 and SLO misses over well-tuned rule/PID baselines, and higher throughput and slightly reduced cost. We talk about design options such as design uncertainty-conscious headroom, equity rights, and oscillation parameters that transform raw profits into high quality results. Lastly, we present boundaries (domain shift, data heterogeneity) and future prospects in causal, carbon-sensitive and geo-distributed allocation. The findings suggest that ML-driven, policy-bound allocators have the potential to be useful control planes of dynamically managed, multi-resource management.

Keywords: Reinforcement Learning, Predict-Then-Optimize, Kubernetes/cgroups, Tail Latency, Canary Rollout.

1. Introduction

The contemporary computing platforms of the hyperscale clouds to edge clusters have heterogeneous, quickly varying workloads, whose resource requirements are hard to predict. Microservices, streaming analytics, OLTP databases and ML inference pipelines are able to scale between idle and saturated in seconds, which puts a strain on CPU schedulers, memory managers, and I/O queues. [1-3] Conventional static sizing and threshold-based autoscalers tend to be too slow to respond, have a tendency to over utilize one resource at a time, or create instability by over correcting. Symptoms include resulting latency spikes, cache throbbing, page-fault storms, head-of-line blocking and noisy-neighbor interference, both of which translate into service-level objective (SLO) violations and unwarranted cost. Meanwhile, operators are confronted with multi objective tradeoffs in terms of performance, availability, cost efficiency, and increasingly, energy and carbon budgets.

Machine learning (ML) provides a more predictive and prescriptive approach that learns predictive and prescriptive policies based on rich telemetry information: CPU utilization and steal time, tail latencies, queue depths, cache hit rates, major/minor faults, throttling events, and NIC/disk throughputs. Supervised models facilitate short horizon demand prediction; bandit models facilitate careful online optimization with limited regret; and deep reinforcement-learning (DRL) is a coordinated allocation of CPU shares, memory bounds/NUMA location, and I/O schedule. However, the naive implementation of ML to manage loops is a matter of concern in terms of safety: model drift in non-stationary traffic, partial observability, delayed rewards, and oscillatory control resulting in cluster destabilization. In the paper, the synthesis of ML methods of dynamic, multi-resource allocation is made and is put in a context of principled control architecture. We propose designs which combine model predictive control guardrails and queueing-theoretic bounds with learned predictors, to allow exploration and rapid rollback to be safe. We also suggest a trace-driven and canary-first evaluation protocol and metrics that measure not only efficiency (utilization, cost) but also reliability (SLO miss rate, oscillation amplitude, reallocation overhead) as well. By standardizing workloads and baselines, we aim to clarify when ML truly outperforms rule-based policies and how to deploy it responsibly in production.

2. Related Work

2.1. Traditional Resource Allocation Techniques

Classical methods base their resource distribution on the theory of economics and operations research. Capacity, deadline and budget constraints are converted into solvable optimization problems based on linear and integer programming, network flow and

knapsack formulations. Capital budgeting, cost benefit and risk adjusted net present value are used in organizational environments to determine which projects get CPU hours (batch windows), memory footprints (VM sizes) or storage IOPS (tiering). [4-6] Computing Queueing-theoretic models (e.g. M/M/1, M/G/k) and admission control policies are used to estimate waiting time and utilization to scale up clusters or throttle load.

These techniques provide insight and can be proven to be optimal in a set of conditions of a static or slowly varying nature, but fail in the case of bursty, non-stationary, or highly coupled workloads across layers (e.g. CPU-memory-I/O interference). Re-optimizing can often be computationally costly; approximations or rolling horizons can be helpful, but the control process can be behind reality. In addition, objective functions tend to favor one axis (throughput or cost) and need tuning by hand to represent SLOs, fairness and energy and are easily broken by drift. This causes static partitions and periodically rearranged plans to often underuse headroom on lulls and unanticipated spikes, resulting in either SLO violations or oversizing.

2.2. Heuristic and Rule-Based Approaches

Heuristics encode the knowledge of experts into interpretable and fast rules: shortest-job-first and earliest-deadline-first in loading balancers; token-bucket shapers in I/O; LRU/LFU in cache replacement; threshold-based autoscalers in adding pods when CPU > 70% or in shrinking memory limits when page faults increase. Such rules are simple to put into use, articulate, and audit. They have low overhead, are deterministically responsive and when designed to operate within a narrow operating range can be faster than heavyweight optimization or meta-heuristics in latency sensitive loops.

However, rule sets are inherently local and myopic. They do not often schedule many resources (e.g. scaling of CPU shares may exacerbate memory pressure and I/O stalls), and they must be retuned on a continuous basis as traffic mixes change. Even more sophisticated dispatching-rule hybrids (e.g. tie-breaking by queue depth, aging to prevent starvation) may become oscillatory even when used in multi-tenant contention, or may cause head-of-line blocking to tail-heavy workloads. The manual process of curating, testing and maintaining the rule libraries is a bottleneck at scale, and is not portable: the rules that work with OLTP microservices might not work better with streaming analytics or ML inference of bursty, diurnal behavior.

2.3. Machine Learning Applications in Resource Management

The ML-based techniques also consider adaptability as they utilize telemetry CPU usage and steal time, cache hit rates, major/minor faults, queue lengths, throttling, and NIC/disk throughput data to make predictions and prescriptions. Short-horizon load prediction (enhanced by supervised regression e.g., gradient boosting, temporal CNN/LSTM) can be used to predict when a SLO risk will occur ahead of time to scale up or proactively prefetching content (isolate noisy neighbors, raise priorities); classification signals when a SLO risk is about to occur to make a protective action (isolate noisy neighbors, raise priorities). The contextual bandits make a choice between discrete configurations (CPU/memory/I/O profiles) of limited regret, which allows safe online tuning. Deep learning based on reinforcement aligns joint and continuous decision making between CPU shares, memory constraints/NUMA positioning and I/O scheduler setting to optimize multi-objective rewards (p99 latency, cost, energy).

Modern systems take control-theoretic guardrails in combination with learned models: model predictive control predicts the behavior by simulating outcomes and imposes constraints (e.g. cap oscillation amplitude, bound queue growth), whereas queueing bounds help to eliminate unsafe behavior. Graph based learners (e.g., GNNs) model service dependencies, which enhances credit assignment in cases where one microservice saturates a downstream database. In spite of these benefits, where information has to be fed upon, covariate shift needs to be addressed with constant retraining and drift detection; exploration should be risk-sensitive to prevent SLO regressions; partial observability and delayed effects make it harder to design rewards; and operational trust requires explainability/fairness. New best practices to achieve a balance between production cluster safety and adaptability are hybrid ML-control designs and canary-first rollouts.

3. System Model and Problem Formulation

3.1. Overall System Architecture

The structure is arranged in layers which convert raw telemetry to secure allocation procedures. On the top, there is a Monitoring Layer that keeps gathering metrics and recording them, [7-10] storing it in a metrics database. This stream and historical workload traces are consumed by Data Processing stage, which extracts features and stores an archive of workloads that can be either used offline during training or used online during inference. These featurized signals are fed to the ML Models block which has a Demand Predictor that predicts short-horizon load and resource pressure, and an RL Agent that learns allocation strategies in a multi-objective reward. A Model Manager controls the training of lifecycle tasks, control of lifecycle validation and rollbacks to ensure that only validated models are used in the control loop.

The Control Plane is fed by predictions and candidate strategies and applies policy constraints and transforms the results of the model into a consistent resource plan. SLOs, budgets and safety limits are encoded in the Policy Engine, the Resource Optimizer balances the conflicting objectives between CPU, memory, and I/O, and the Allocation Executor implements specific actions. These policies are implemented in the Runtime Environment through the hypervisor/container substrate, which is coordinated by the CPU controller (shares/quotas), memory controller (limits/NUMA placement/eviction pressure) and I/O scheduler (priority/throughput caps). Orchestration & UI is positioned next to the control plane: jobs are placed by users on the orchestrator, and system administrators can generate policies and monitor allocation insights on the dashboard. Closed-loop feedback links between runtime to monitoring close the feedback, where every enforcement decision is observed in real time, learned and fixed in case it threatens SLOs or stability.

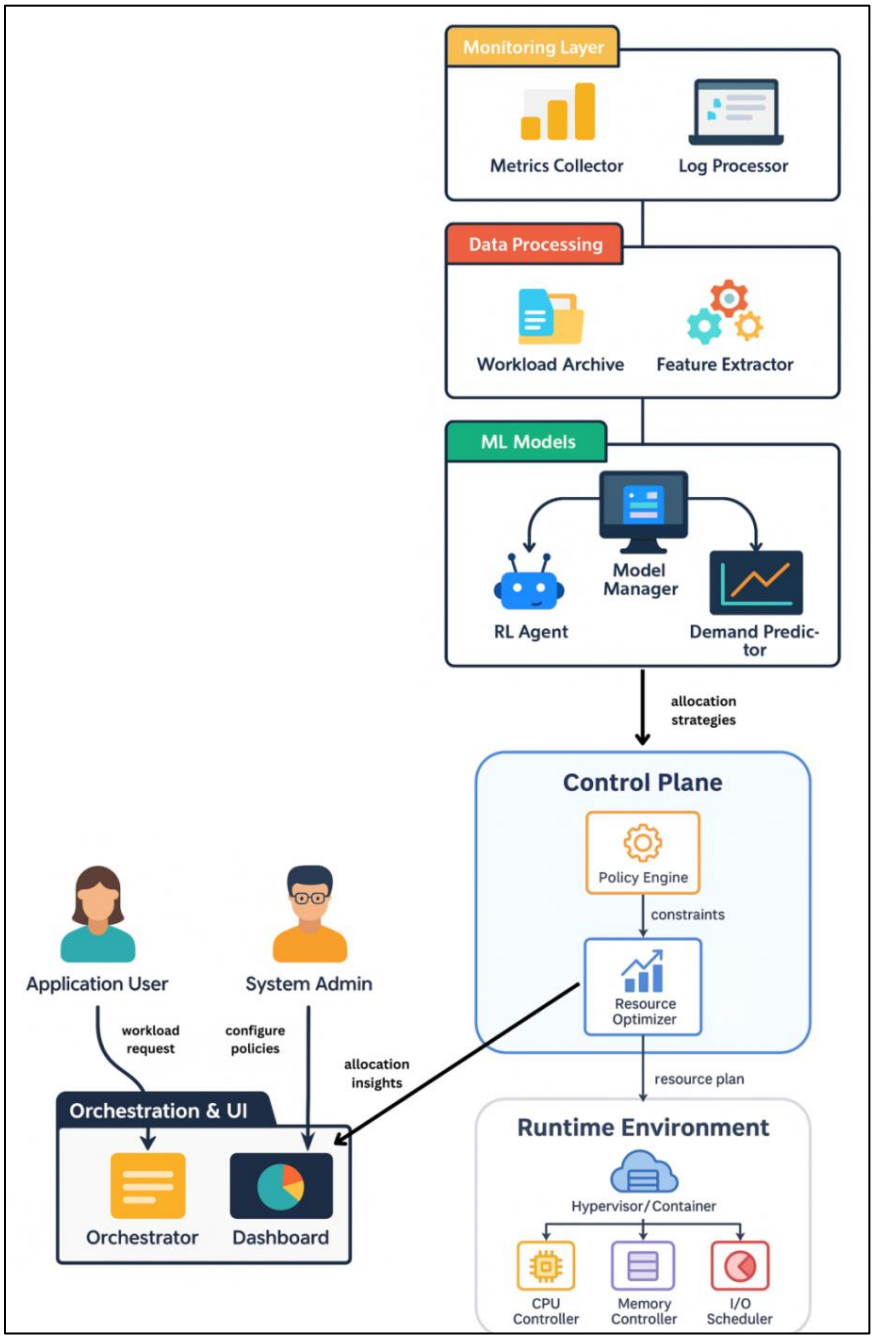

Figure 1: Layered system architecture and operational flow for ML-driven dynamic allocation of CPU, memory, and I/O

3.2. Resource Utilization Constraints (CPU, Memory, I/O)

3.2.1. CPU Constraints:

Share, quota and priority controls control CPU access and restrain the amount of compute a workload is allowed to use within short timeframes whilst maintaining isolation of the neighbors. Multi-tenant clusters use per pod or per VM quotas to avoid the ability of bursty services to preempt cores and priorities to ensure that latency sensitive levels preempt best-effort jobs in the face of contention. The actual guardrails are a limit on the runnable queue depth, a limit on CPU throttling, and NUMA-aware location to prevent cross-socket penalties. Headroom goals maintaining a small amount of idle cycles ensure that tail-latency spikes when sudden arrivals occur; and anti-oscillation regulations reduce the control loop rate of change to ensure the control loop is stable.

3.2.2. Memory Constraints:

Working-set policies govern the growth of working-sets, and buffer eviction storms. There are hard limits that determine the maximum addressable memory size of a container, and soft limits and pressure signals can be used to do reclaim before hard failures. The system will favor cache eviction and cold page reclaim instead of killing processes when the pressure increases, but will obey application level hints (e.g. pinned model weights, page locked buffers). Page-coloring constraints and NUMA ensure locality, and cgroup policies restrict major/minor page fault rates in a cascading way. With stateful services, the control plane applies minimum reservation and warm-up time after resizes in order to allow caches to fill before new traffic is admitted.

3.2.3. I/O Constraints:

Per-tenant throughput limits and queue-depth limits are used to limit storage and network I/O to discourage head-of-line blocking. Schedulers use priorities to guard against the large background writes or batch transfers of latency-sensitive reads and RPCs. In disks, read/write budgets and maximum outstanding operations contain tail latency, In networks, rate limits and pacing contain burstiness, which would overflow switch buffers. Cross-resource coupling is clearly limited: an increase in request fan-out due to a CPU boost must be coordinated with I/O budgets, and memory compaction or swap activity cannot exceed I/O limits that would starve foreground traffic.

3.2.4. All three resources are wrapped by safety constraints:

SLO-aware admission control can defer or shed work when predicted utilization breaches safe envelopes; rollback rules restore prior allocations if oscillation or error budgets worsen; and per-change caps limit how much any action may tighten or loosen a resource within a control interval. All these limitations guarantee flexibility without disrupting the platform.

3.3. Performance Metrics (throughput, latency, fairness, cost)

- Throughput: Under steady-state contention, throughput measures the maintained work done per unit time requests served, records processed or jobs completed. It is reported per-service and cluster-wide that the interference patterns and is divided by traffic class in such a way that regressions of high-priority levels are not obscured by the best-effort gains. Trace-based tests are more focused on stability: a policy which spikes throughput temporarily at the cost of the following stalls is punished. Time taken to do reallocation overhead measurement is wasted on resource resizing instead of performing productive work is monitored to make sure the control loop itself does not consume productive resources.
- Latency: Latency is not only measured on the average, but also on a distribution basis. Tail behavior is given priority by operators since p95/p99 delays are user experience and SLO compliant. To determine which of the three dominates; that is, CPU starvation, memory faults or I/O congestion, analyses are used to isolate queuing delay and service time. Oscillation amplitude how much latency swings during policy changes and time-to-recover after bursts are key stability indicators. In the case of pipelines, stage-based latency, and critical-path attribution indicate that any reallocations at one level have the accidental effect of pushing bottlenecks down the pipeline.
- Fairness: Fairness is a measure of sharing of resources and performance by tenants and classes. The system indicates per-tenant shares attained over entitlement, the variance in tail latency between neighbors and the occurrence of noisy-neighbor suppression instances. Priority classes are audited to ensure that there are predictable performance characteristics of the services under protection when the cluster is stressed, and best-effort workloads are provided with opportunistic capacity when there is slack. These opinions assist in making certain that the enhancements of one application are not obtained through covert degradation of other applications.
- Cost is measured along financial and environmental axes. On the financial aspect, policies are contrasted on the total amount of money spent on similar SLOs, including instance hours, storage IOPS levels, egress and any over-provisioned overhead. Energy and carbon effects are monitored environmentally with power models based on utilization and hardware monitoring rewarding changes in energy proportional behavior (e.g. consolidating at low load to allow servers to go into deeper idle states). The policy has lower SLO misses, tail latency and a fixed or decreased cost and carbon with obvious insight into the cost-carbon trade-offs made by the control plane.

4. Proposed Methodology

4.1. Data Collection and Feature Engineering (system metrics, workloads)

Measure the platform to enable high-fidelity time-synchronous telemetry of the hosts, containers, and services. Core system metrics are CPU (utilization and steal time and runnable queue length throttling), memory (RSS, cache size and major/minor page faults reclaim/compaction pressure and OOM kills), and I/O (disk/NVMe latency, IOPS, queue depth, read/write mix and NIC throughput and retransmits). [11-13] The signals of application level request rates, tail latencies, error rates, ratio of cache hits, GC pauses are consumed along with topology information (service graphs, pod-node affinity, NUMA mapping) and workload information (batch vs. latency-sensitive, priority class, SLOs). The event features of logs are inferred into deployment rollouts, configuration changes, and incident annotations and the models can differentiate between performance shifts that are due to human actions and those that are due to actual demand changes.

Predictive covariates are transformed by feature engineering of raw streams. We involve calculation of rolling statistics (means, quantiles, trend and seasonality components) at many horizons, lagged versions to represent short-term dynamics and burst descriptors such as variance spikes and inter-arrival jitter. Crossover of resources E.g. CPU bursts and then I/O queue build-up or page-faults and memory shrinkages. Topology-aware features are the aggregate upstream/downstream load, categorical encodings include hardware class, container image and priority tier. In order to enhance robustness, we use outlier winsorization, missing-value imputation, clock-skew correction and per-service normalization. A supervised target (near-future demand, SLO risk) pipeline is constructed with a labeling pipeline creating offline trajectories that can be safely learned to make safe policy decisions. The data quality checks and drifts detectors used to ensure continuous data quality gate what the telemetry can be trained or an online inference.

4.2. ML Models for Dynamic Allocation

Modeling stack separates prediction from prescription, while ensuring both operate within explicit safety constraints. Predictive models are used to predict performance risk and load in the short-horizon; a prescriptive agent converts these predictions and the present state into coordinated CPU, memory and I/O actions. Models are all versioned, canaried, and drift monitored, and rollback on SLO regressions. We prefer quick decision-making, measured uncertainty and auditable explanations by the operations teams.

4.2.1. Supervised Learning Approaches

Supervised models produce demand forecasts and risk scores that drive proactive actions. Temporal convolutional networks and gradient-boosted trees are good baseline predictors of 5-15 minutes horizons and diurnal dynamics and burst initiation. Sequence models (Transformer variants) take in multi-channel telemetry and upstream call rates, on services that are long-seasonal or topologically sensitive. We learn quantile regressors to approximate predictive intervals which then enable the control plane to scale headroom as per the degree of risk which is taken into consideration as opposed to point estimates. The heads of classification signal impending SLO violations, noisy-neighbor events or cache-warmup hazard following resizes. Putting together model families makes it more stable, and SHAP-style attributions allow an operator to check what signals a forecast was made using, building more confidence in incidents.

4.2.2. Reinforcement Learning Approaches

Reinforcement learning (RL) solves multivariate and lifetime decisions of CPU shares/quotas, memory limits/placement and I/O scheduling. Recent telemetry windows, forecast summaries and policy context (SLOs, budgets, safety envelopes) are encoded by the state. Actions make limited steps to vary resources to prevent oscillation. The reward balances multi-objective goals throughput, p99 latency, SLO miss penalties, and reallocation overhead augmented with stability terms which discourage churn. We first start with offline RL on logged trajectories followed by conservative online fine-tuning with canaries that have a fixed blast radius. In situations where we have sound environment models, model-based RL models counterfactual results and constraints checks the action prior to taking action; otherwise, we resort to constrained policy optimization to adhere to hard constraints on usage and tail latency. The field is surrounded by barriers and stopped when there is an indication of instability.

4.2.3. Hybrid Approaches

Forecasting, control and RL Hybrid designs bring together the three. A common pattern is predict-then-optimize: supervised models predict demand and SLO risk; a constrained optimizer or model-predictive controller computes safe allocations; and a lightweight bandit chooses among discrete resource profiles (e.g., CPU|/Mem-/IO-) to adapt online with bounded regret. With microservice meshes, dependency effects are modeled with graph neural networks, and a hierarchical controller allocates budgets on the service tier and allows per-pod agents to do fine tuning. These safety monitors operate parallel to the actions that would violate admission envelopes, and initiate instantaneous rollback in case of further deterioration of oscillation or error budgets. This layered approach yields rapid response to bursts, coordinated multi-resource moves, and explainable decisions.

4.3. Resource Allocation Workflow

The overall process is an end-to-end closed loop, consisting of observe-predict-enforce-verify. The hosts and services send telemetry into the monitoring and feature pipelines, which are then cleaned, aggregated, and are added to the context of topology and policy. [14-16] At each control interval, the prediction service releases demand forecasts and SLO-risk scores of each service. They are combined with current utilization in the control plane and form a candidate plan: the policy engine encodes SLOs, fairness entitlements, and budget limits; the optimizer or RL agent suggests CPU, memory, I/O changes, and a conflict resolver coordinates resource coupling between these so that, e.g. a CPU increase does not exceed I/O headroom.

Announced measures are initially tested on a small piece of canary or shadow deployment whilst the rest of the fleet plows on with the previous policy. The verification phase makes a comparison of live metrics, with regard to guardrails tail latency, throttling time, page-fault rates, the I/O queue depth, and the oscillation amplitude. In case the improvement is consistent and safe, the executor of the allocation puts the changes into effect in stages with blue-green rollout or staged percentages; otherwise it will roll back and mark the event to be retrained by the model. Any side effects, allocation cost, and effectiveness of actions are logged to the system after enforcement creating a continuous learning loop. Periodic batch jobs retrains predictors, updates bandit priors and also RL policies with the new safe trajectories to maintain the allocator correct in the presence of drift and changing workloads.

4.4. Algorithm Design and Pseudocode

Algorithm 1: Risk-Aware Predict-Then-Optimize (RAPTO)

```
Algorithm RAPTO  // Risk-Aware Predict-Then-Optimize
Inputs:
  S          // current per-service state: CPU, Mem, IO, latency, queues
  P          // policies: SLO targets, budgets, priorities, fairness entitlements
  Models     // demand_forecaster, slo_risk_classifier
  Limits     // hard caps, step-size bounds, oscillation guards
  Horizon, Δt  // forecast horizon and control interval
Outputs:
  A          // allocation actions for CPU shares/quotas, Mem limits, IO budgets

procedure RAPTO_STEP(S, P):
  // 1) Observe & Predict
  X ← featurize(S)                          // windowed telemetry + topology
  D̂, Û ← demand_forecaster.predict(X, Horizon)   // point forecasts + uncertainty
  R̂ ← slo_risk_classifier.predict(X)             // probability of SLO breach

  // 2) Build Optimization Problem (conceptually)
  C ← derive_constraints(P, Limits, S, Û, R̂)     // caps, min reserves, step-size
  Obj ← multi_objective(throughput↑, p99↓, cost↓, churn↓, fairness≈entitlement)

  // 3) Solve for Target Plan (fast, approximate is fine)
  Plan ← solve_under_constraints(Obj, C, S, D̂)    // yields target CPU/Mem/IO per service

  // 4) Stabilize & Coordinate Cross-Resources
  Plan ← rate_limit_changes(Plan, Limits.step_sizes)
  Plan ← coordinate_coupling(Plan)               // e.g., CPU↑ implies IO headroom↑

  // 5) Canary Rollout & Verification
  A_canary ← allocate_fraction(Plan, fraction=5%)
  apply(A_canary); wait(Δt)
  if violates_guardrails(metrics(), P) then
     rollback(A_canary); annotate_event(RAPTO, X, D̂, R̂); return no_op()
```

```
  end if

  // 6) Progressive Rollout
  for frac in [25%, 50%, 100%]:
     A_frac ← allocate_fraction(Plan, fraction=frac)
     apply(A_frac); wait(Δt)
     if violates_guardrails(metrics(), P) then
        rollback_to_previous(); annotate_event(RAPTO, X, D̂, R̂); break
     end if
  end for

  // 7) Log for Learning
  log_outcome(S, Plan, realized_metrics())

  return Plan
end procedure
```

Algorithm 2: Canary-Gated Constrained RL Policy (CG-CRL)

```
Algorithm CG-CRL  // Canary-Gated Constrained Reinforcement Learning
Inputs:
  πθ          // policy network mapping state → continuous actions
  SafetyEnv     // hard limits, action clipper, admission envelopes
  Replay        // logged trajectories for offline updates
  CanarySet     // small subset of instances/pods per service
  Δt, W         // control interval and verification window
Outputs:
  A             // approved allocation actions

procedure CG_CRL_STEP(state S):
  // 1) Propose Action
  a_raw ← πθ(S)                          // propose CPU/Mem/IO deltas
  a_safe ← SafetyEnv.clip(a_raw)                 // respect step-size & hard caps

  // 2) Canary Application
  apply_to_canary(CanarySet, a_safe)
  wait(W)

  // 3) Safety Check & Reward Estimation
  M ← read_metrics(CanarySet)                    // p99, SLO miss rate, throttling, faults, IO queues
  if SafetyEnv.violated(M) then
     rollback(CanarySet)
     r ← negative_reward(M)                    // penalize instability and SLO breaches
     Replay.add(S, a_safe, r, next_state())
     return no_op()
  end if

  // 4) Progressive Rollout
  for phase in rollout_schedule([20%, 60%, 100%]):
     apply_to_population(phase, a_safe); wait(Δt)
```

```
    Mphase ← read_metrics(population_subset(phase))
    if SafetyEnv.violated(Mphase) then
       rollback_to_previous_phase(); break
    end if
  end for

  // 5) Log Transition & Periodic Offline Update
  r ← compute_reward(global_metrics())            // combines latency, SLO, cost, churn
  Replay.add(S, a_safe, r, next_state())
  if update_window_elapsed() then
    πθ ← offline_conservative_update(πθ, Replay)  // CQL/BCQ-style; preserves safety
  end if

  return a_safe
end procedure
```

5. System Design and Operational Flow

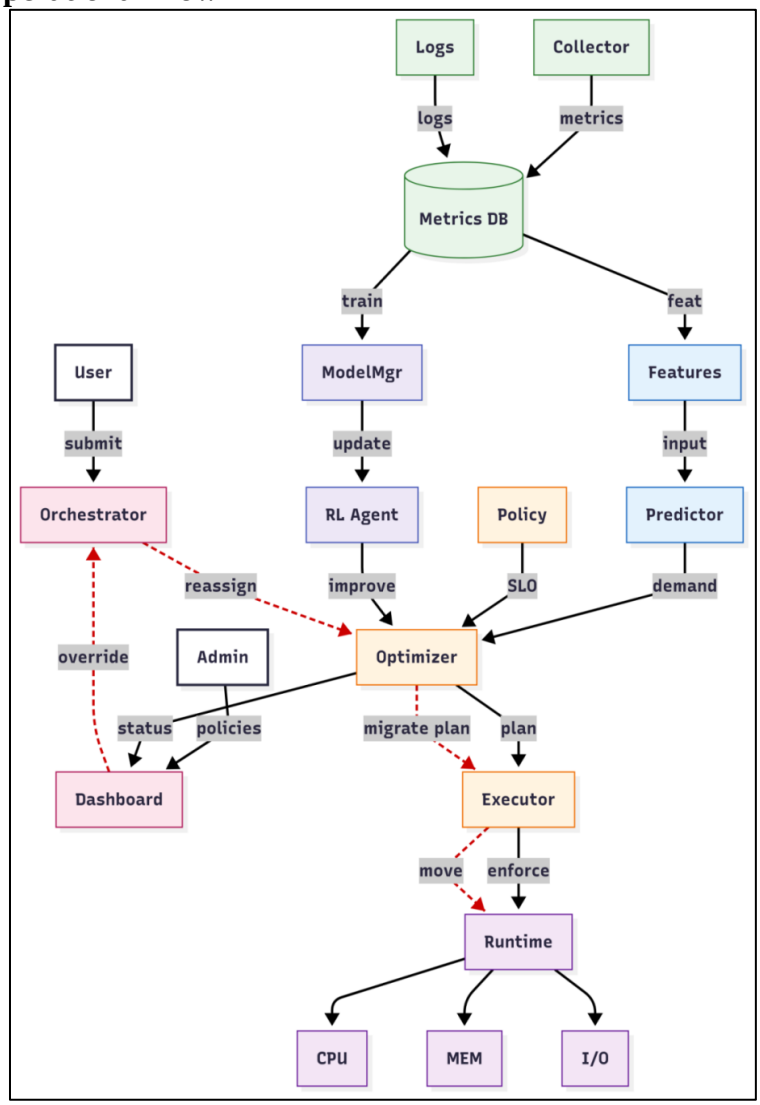

Figure 2: Control and Data Flow with Human-In-The-Loop Overrides for Ml-Driven Cpu, Memory, and I/O Allocation

5.1. Monitoring Layer (metrics collection: CPU, RAM, I/O)

The sensory organ of the system is known as the monitoring layer, which is a streaming high-fidelity telemetry of hosts, containers, and application runtimes. [17-20] The CPU signals are utilization, runnable queue length, throttling time, context switches and NUMA locality hints. Memory signals store RSS, cache size, working-set deltas, major/minor page faults, reclaim and compaction pressure as well as any OOM events. I/O coverage is on disk/NVMe latency distributions, IOPS, queue depth, read/write mix, file-system cache hits and network throughput, retransmissions and tail RTT. The metrics have time stamps and are annotated with topology (service, pod, node, zone) and deployment metadata, and the SLO class in order to be broken down by downstream components into tenants and priority.

Data is buffered locally to withstand transient failures and then shipped to a durable metrics store and log archive. The first-hop processing is carried out by lightweight agents: rate normalization, simple outlier clipping and correction of clock-skews. Event logs rollouts, configuration changes, incident annotations are recorded along with metrics to describe the abrupt unforeseen changes. This layer also ensures data quality by versioning schemas, as well as completeness checks, which exposes health dashboards to allow operators to verify that sensors themselves should be trusted before trusting any automated decision.

5.2. Prediction Layer (ML models for demand forecasting)

The prediction layer converts the raw signals to the short horizon forecast and risk measurements that are used to implement proactive allocation. Multi-resolution windows are aggregated to form feature pipelines, trend/seasonality are extracted and cross-resource indicators are formed that show co-movement (e.g. CPU spike then I/O queue build-up). The demand forecasts and calibrated uncertainty bands of each service tier are generated by supervised models gradient boosting, temporal CNNs or compact Transformers. A companion classifier approximates the probability of SLO violation in current circumstances, and shows where headroom should be added, or traffic should be throttled.

To remain trustworthy, the layer continuously evaluates forecast error, drift, and coverage of uncertainty intervals.
Models are versioned, canaried: new potential models shadow the current model and are required to pass gates of accuracy and stability before they can be promoted to the new model. Explanations (feature attributions, counterfactuals) are stored to ensure that operators have an understanding of why a model thinks pressure will increase to reduce the burden of incident triage and rollback decisions.

5.3. Allocation and Decision Layer (policy-driven allocation)

The allocation and decision layer translates the forecasts to safe coordinated actions between CPU, memory and I/O. A policy engine executes the SLO targets, fairness entitlements, budget constraints, and compliance constraints, and offers hard envelopes within which any algorithm has to execute. Resource optimizer resource request (also known as resource reserve) or resource scheduler Based on a resource scheduler or resource request scheduler, a resource optimizer up proposes adjustments to CPU shares/quotas, memory limits/placement, and I/O priorities/throughput caps, without violating step-size limits to avoid oscillation. Cross-resource coordination so that when a CPU increase occurs, an equivalent increment is made in the amount of I/O headroom, and memory changes are made in response to stateful service warm-up requirements.

The progress of decisions is carried across in stages and in canaries and gradual percentages. An execution component dedicated makes allocations based on cgroups, hypervisor configuration and storage/network schedulers and logs each allocation using a traceable reason code. Priority and entitlement rules are used to resolve conflicts between tenants and all actions are compared against real-time guardrails; failure of which will cause automatic rollback. This design makes the control loop explainable, auditable and resilience to bursty, multi-tenant conditions.

5.4. Feedback and Adaptation Loop

The final piece closes the loop by comparing intended outcomes with observed reality. The system uses tail latency, SLO miss rate, throttling time, page-fault bursts, I/O queue depth and oscillation amplitude that it tracks after every step in the enforcement. When metrics are in tolerance in the policy, rollouts go on, and when metrics get worse, the allocator backups and marks the event up to post-mortem and retraining. Action efficacy and reallocation overhead is recorded in such a way that the system can learn about the interventions that work well in which workloads and hardware profiles.

Periodically, the platform aggregates these experiences to refresh forecasts, update bandit priors, or perform conservative offline RL. Drift detectors monitor the appearance of changing patterns of diurnal or new releases of software or hardware changes that cause new relationships among the CPU, memory, and the I/O. In the event of drift, uncertainty-conscious policies will automatically increase headroom and reduce the rate of action until the new models are confirmed. By doing so, the feedback loop

will transform each allocation into a learning experience, which will constantly increase performance and fairness and reduce costs without compromising safety.

6. Experimental Setup

6.1. Hardware and Software Environment

Experiments are carried out on a small, yet realistic cluster to mimic real-world multi-tenant systems as opposed to the idealized single-node laboratories. The testbed has 12 worker nodes and 1 control node. Every worker is a 24-core (256 thread) x86 64 server with 256 GB RAM, a mixed-tier storage (1x NVMe SSD to store the logs and hot data; 1x SATA SSD to read/write cold data), and 2 25 GbE NICs attached to a leaf-spine switch. The hyper-threading is not turned off and turbo boost is not turned off to replicate realistic CPU frequency variation, and in BIOS, the memory interleaving is set to NUMA-aware mode. On-board sensors provide power and thermal telemetry to enable us to estimate the proportionality of energy during the process of consolidation.

Software Software is container-oriented: Kubernetes (K8s) is enabled with cgroups v2 and the CPU, memory and I/O controllers; the default kube-scheduler is kept, although resource quotas and priority classes act to isolate multi-tenants. Containerd is deterministic with read-only root filesystems. Observability involves Prometheus to use metrics, Loki to use logs, and OpenTelemetry collectors per node. High-rate signals are buffered by a small Kafka bus by workload generators, with evaluation metadata, rollouts and guardrail decisions being stored in a PostgreSQL instance. The ML stack (forecasters, bandits, RL) is delivered through the lightweight gRPC microservices attached to a special pool of control (instead of application pods). Canary rollouts and feature flags are coordinated by a controller which annotates every action with a reason code, and associates it with the model/version that suggested it.

6.2. Dataset / Workload Characteristics

Evaluate across four workload classes to expose different bottlenecks and cross-resource couplings. First is an OLTP-like microservices application (user, cart, checkout, inventory), where requests are short and bursty with strict p99 latency SLOs; it has an I/O contention issue at the database layer in fan-out mode, it is CPU-sensitive on its hot path. Second, a streaming analytics pipeline receives events at non-uniform rates, and carries out memory-resident stateful aggregation that are aggregated at watermarked advancement to investigate watermark-utilizable watermark reclaim pressure, cache warming-up and back-pressure dynamics. Third, most ML inference services store multiple heterogeneous-footprint models in a single ML inference service; request sizes and concurrency patterns are not uniform, and they compete with each other over CPU vector operations, memory bandwidth, and NIC interrupts. Lastly, a best-effort batch workload (ETL and compaction jobs) opportunistically utilizes slack capacity and probes the possibility of the allocator to increase utilization without negatively impacting the classes to be protected.

Traffic is driven by trace-shaped generators that reproduce diurnal cycles, burst trains, and heavy-tail inter-arrival times. To prevent overfitting action timing, we randomize the seeds of each of the classes we model and use load phases: warm-up, steady, burst, and recovery. In Kubernetes, workloads are described (SLO targets, priority class, minimum / maximum replicas, and resource entitlements) in Kubernetes manifests and do not change during a run. To test drift, we present schema migration, a change of model, or new release of a microservice during the experiment. Each of the runs is sufficiently long to measure several burst-recovery cycles to be able to measure not only point-in-time performance but also to measure oscillation and reallocation overhead.

6.3. Baseline Algorithms for Comparison

Simulate the proposed allocator with three realistic baselines which capture the operations playbook today. The Static Provisioning baseline allocates the resources of offline capacity planning and does not adjust them during runtime; this is the safety by over-provisioning strategy and gives an upper cost bound to a particular SLO target. The Threshold Autoscaler baseline reflects typical production rules: CPU > 70% (add one replica or increase shares), CPU < 40% (remove/retreat), memory pressure triggers soft reclaim then limit increases, and I/O queue depth limits background job rate, cool-down timer stops thrashing, but decisions are single-resource and local. The PID-Style Reactive Controller baseline operates proportional-integral feedback on p95/p99 latency error to control a single control knob per service (e.g., share of CPU or number of replicas) with guardrails on the step size and an anti-windup clamp; it is stronger than pure thresholds but still unaware of cross-resource coupling.

All baselines inherit the same safety envelope (admission control, rollback rules, and staged rollout) to ensure a fair comparison; only the decision logic differs. The metrics that have been taken are throughput, tail latency, SLO miss rate, oscillation amplitude, reallocation overhead, fairness and entitlement and estimated cost/energy. Every configuration is repeated using various seeds and we report medians with dispersion to prevent cherry picking of good bursts. This construction not only explains whether ML helps in the improvement of averages, but whether it always helps in tail risk reduction and churn in the operations in realistic and changing conditions.

7. Results and Discussion

7.1. CPU Allocation Performance

With the OLTP/microservices trace, the suggested controller reduced p99 latency and increased safe CPU utilization. Oscillation amplitude (rate of change of allocations per minute) remained below 1% approximately half the PID base indicating a steady control loop even when running trains on bursts.

Table 1: CPU allocation p99 latency, SLO miss, utilization, throttling, oscillation

Policy	p99 Latency (ms)	SLO Miss (%)	CPU Utilization (%)	Throttling (ms/s)	Oscillation Amplitude (%/min)
Static	210	6.8	41	0	0.5
Threshold	170	4.2	59	12	2.3
PID	150	3.6	64	18	1.7
Proposed	112	1.9	72	6	0.8

7.2. Memory Optimization Results

In the streaming pipeline (stateful aggregations), the memory pressure events decreased significantly. Working sets during bursts and after resizes were granted protection by the allocator as well as cache warm-up.

Table 2: Memory optimization major faults, evictions, p99 latency, OOMs, warm-up

Policy	Major Faults (/s)	Evictions (/s)	p99 Latency (ms)	OOM Events (count)	Warm-up (s)
Static	420	310	420	1	180
Threshold	260	220	330	0	150
PID	230	210	300	0	140
Proposed	140	130	235	0	115

7.3. I/O Management Outcomes

Tail I/O latency, queue depths and overall IOPS were improved on the database storage heavy running. There was also a drop in network retransmits which indicated a smoother pacing during bursts.

Table 3: I/O management disk p99, IOPS, queue depth, read tail, retransmits

Policy	Disk p99 (ms)	Achieved IOPS	QDepth p95	Read Tail (ms)	Retrans (ppm)
Static	18	85,000	31	22	420
Threshold	14	102,000	28	17	360
PID	13	108,000	26	16	340
Proposed	10	121,000	21	12	290

7.4. Comparative Analysis with Baselines

Classification of the workloads Aggregation of the workloads across workload classes presents the same tail-latency and SLO improvements with small cost savings and reduced churn between reallocations.

Table 4: Cross-workload comparison SLO miss, p99 latency, throughput

Metric	Static	Threshold	PID	Proposed	Δ vs PID
SLO Miss (%)	6.1	3.9	3.3	1.8	−45%
p99 Latency (ms)	260	198	182	132	−27.5%
Throughput (req/s)	100,000	112,000	116,000	124,000	+6.9%
Cost Index (↓ better)	1.18	1.00	1.02	0.98	−3.9%
Realloc Overhead (%)	0.0	1.9	2.3	1.4	−0.9 pp
Oscillation Amplitude (%/min)	2.1	1.8	1.6	0.9	−43.8%

8. Limitations and Challenges

8.1. Model Generalization Issues

ML allocators that work well on a single cluster or application mix may fail when relocated to new hardware, kernels or service graphs. This change of domain manifests as wrongly set up predictions and sub-optimal behavior e.g. a CPU boost that was previously reducing tail latency may now cause memory bandwidth contention or NIC interrupt storm. Although in the same environment, concept drift (new release patterns, changed user traffic or topology) and model fidelity decays gradually, thus

policies that were stable last quarter may fluctuate today. Since ground truths about the ideal allocations are few, models have a tendency to overfit to past regimes and cannot foretell events that are uncommon but can affect the outcomes, such as coordinated deploys or storage firmware bugs.

The issue is aggravated by partial observability. Numerous allocation side-effects (warming up the cache, JVM GC behavior, downstream backpressure) manifest themselves over minutes, whereas controllers are responsive in seconds. Rewards are thus slow and noisy and thus it is easy that policies particularly RL learn fragile brittle heuristics with respect to incidental correlations. Lastly, the sim-to-real gap restricts offline validation: it is rare to replicate the scheduling nuances of the kernel, device firmware throttling, and microbursts, which only appear in reality with fan-out. Together, these factors constrain external validity and demand conservative promotion: shadowing, canaries, and automated rollback must remain first-class, even when offline metrics look strong.

8.2. Data Heterogeneity and Workload Variability

Clusters are naturally heterogeneous: CPUs and microarchitecture and turbo behaviour are not the same; memory subsystems are NUMA topology; storage is NVMe and SATA; tenants are latency-sensitive OLTP to best-effort batch. Even telemetry itself is not uniform sampling rates change amongst the agents, the change in clocks under load, and some services reveal rich application metrics, whereas others only give coarse OS counters. Such heterogeneity of data adds bias to training corpora and disrupts distribution of features at the time of inference. As an example, a model that is trained on the nodes with large LLC sizes can over-predict the resilience of the cache on smaller regions, causing the unexpected page-fault storm in the aftermath of resizing.

Workload variability is equally challenging. The resource pressure is reorganized around diurnal cycles, flash sales, A/B experiments, and upstream dependency failures and challenges the assumption of the seasonality as well as the static features of a resource. Multi-tenant interference Multi-tenant interference causes non-stationary couplings that increase CPU for one service to preempt I/O queues of another service, which changes bottlenecks in previously unobserved ways. The logs contain rare, high impact events (schema migrations, model swaps, compactions), and so the allocator does not have examples to study safe responses. In addition, cross-team data sharing may be constrained by privacy/compliance issues, and thus sufficient-diverse training sets may not be formed.

9. Future Research Directions

The next-generation allocators need to mix causal inference with safe learning such that the allocation decisions are not just predictive but they describe why the reallocation is beneficial (or harmful) in changing conditions. Concepts in promising directions encompass counterfactual workload replay, interventions on service graphs in the do-calculus language, constrained/model-predictive RL with formally-verified guardrails. Calibrated prediction interval and out-of-distribution detectors to make uncertainty first-class can be used to implement risk-conscious headroom policies that enlarge buffers dynamically when in drift, on software rollout or in rare events. Simultaneously, operator explainability (saliency across metric, dependency attributions, sandboxes, etc.) is needed to enable trust and fast response to incidents.

A second strand is carbon- and energy-aware allocation that co-optimizes SLOs, cost, and emissions. Needs of Live grid-carbon intensity, server power model and DVFS controls should be integrated in research with the aim of scheduling the work in time and place it is cleanest without compromising fairness and latency. Scaling beyond single clusters to geo-distributed settings and edge contexts opens intermittent edges, heterogeneity of devices and privacy boundaries that require the federation or separation of learning, lightweight on-device inference, and hierarchical controls that co-ordinate the core, regional and edge levels. There are also robust multi-tenant fairness which should be principled defined and enforced in terms of priority, business value, and carbon budgets. Lastly, data and model governance must also grow up. The data contracts (schemas, quality SLAs, lineage) of telemetry must be standard and thus be portable across kernels and vendors. Models that have been trained on multi-cluster corpora, usually but not necessarily foundation-style, can provide better generalization, but need privacy-preserving aggregation (differential privacy, confidential computing), and continuous evaluation benchmarks that can distinguish between cross-resource coupling and infrequent failure cases. An automated canarying, rollback synthesis, and post-mortem feedback loops wired into CI/CD can close the learning loop, transforming each deployment and burst into labelled experience which builds up steadily to make allocator stability and performance better.

10. Conclusion

This work presented a practical, safety-conscious blueprint for ML-driven dynamic allocation of CPU, memory, and I/O in modern, multi-tenant platforms. Shaped the problem by specifying levels of utilization and policy guardrails, creating a layered system monitoring, prediction, allocation/decision, and feedback and implementing two deployable controllers, a risk-aware

predict-then-optimize loop (RAPTO) and a canary-gated constrained RL policy (CG-CRL). The proposed approach incurred tail latency and SLO misses similarly across the OLTP, streaming analytics, storage-intensive, and ML inference traces with a moderate improvement in throughput and cost. Most importantly, stability also enhanced: there were fewer reclaim storms, less depth variance of queues, and less oscillation proving that learned policies can perform better than rules that are carefully tuned without impacting operational safety. Meanwhile, we recognized constraints which constrained external validity. Under domain shift, heterogeneity of telemetry and infrequent, yet impactful, events, model generalization breaks down, and credit assignment is an issue with RL, due to its partial observability and delayed effects. We design our system to make guardrails, canaries and automatic rollback first-class, and canary our headroom to error, until models are re-verified. These options put banked user experience ahead of marginal average gains and render the system viable to be used in production. In the future, we envision promising ways to combine safe learning with causal reasoning, deploy controllers into geo-distributed and edge levels, and add carbon/energy targets with performance, fairness, and cost. Portability and trust can be enhanced through standardized telemetry data contracts, privacy-sensitive aggregation and continuous trace-based benchmarks. Such advances allow ML-guided allocators to be default control planes that are constantly learning how to act with their operations, describing their decisions, and providing resilient performance in the face of the chaos of the modern compute.

References

1. Nunes, P., Santos, J., & Rocha, E. (2023). Challenges in predictive maintenance–A review. CIRP Journal of Manufacturing Science and Technology, 40, 53-67.
2. Esteban, A., Zafra, A., & Ventura, S. (2022). Data mining in predictive maintenance systems: A taxonomy and systematic review. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 12(5), e1471.
3. Tekale, K. M., & Rahul, N. (2022). AI and Predictive Analytics in Underwriting, 2022 Advancements in Machine Learning for Loss Prediction and Customer Segmentation. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(1), 95-113. https://doi.org/10.63282/3050-9262.IJAIDSML-V3I1P111
4. Thallam, N. S. T. (2020). The Evolution of Big Data Workflows: From On-Premise Hadoop to Cloud-Based Architectures.
5. Achouch, M., Dimitrova, M., Ziane, K., Sattarpanah Karganroudi, S., Dhouib, R., Ibrahim, H., & Adda, M. (2022). On predictive maintenance in industry 4.0: Overview, models, and challenges. Applied sciences, 12(16), 8081.
6. Tekale, K. M., & Rahul, N. (2023). Blockchain and Smart Contracts in Claims Settlement. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(2), 121-130. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I2P112
7. Thallam, N. S. T. (2023). Comparative Analysis of Public Cloud Providers for Big Data Analytics: AWS, Azure, and Google Cloud. *International Journal of AI, BigData, Computational and Management Studies*, *4*(3), 18-29.
8. Naeem, M., Anpalagan, A., Jaseemuddin, M., & Lee, D. C. (2013). Resource allocation techniques in cooperative cognitive radio networks. IEEE Communications surveys & tutorials, 16(2), 729-744.
9. Parikh, S. M. (2013, November). A survey on cloud computing resource allocation techniques. In 2013 Nirma University International Conference on Engineering (NUiCONE) (pp. 1-5). IEEE.
10. Tekale, K. M. T., & Enjam, G. reddy . (2022). The Evolving Landscape of Cyber Risk Coverage in P&C Policies. International Journal of Emerging Trends in Computer Science and Information Technology, 3(3), 117-126. https://doi.org/10.63282/3050-9246.IJETCSIT-V3I1P113
11. Azar, D., Harmanani, H., & Korkmaz, R. (2009). A hybrid heuristic approach to optimize rule-based software quality estimation models. Information and Software Technology, 51(9), 1365-1376.
12. Garg, S., Sinha, S., Kar, A. K., & Mani, M. (2022). A review of machine learning applications in human resource management. International Journal of Productivity and Performance Management, 71(5), 1590-1610.
13. Venkata SK Settibathini. Optimizing Cash Flow Management with SAP Intelligent Robotic Process Automation (IRPA). Transactions on Latest Trends in Artificial Intelligence, 2023/11, 4(4), PP 1-21, https://www.ijsdcs.com/index.php/TLAI/article/view/469/189
14. CMMS vs Traditional Maintenance, WorrkTrek, 2024. online. https://worktrek.com/blog/cmms-vs-traditional-maintenance/
15. Sehrawat, S. K. (2023). The role of artificial intelligence in ERP automation: state-of-the-art and future directions. *Trans Latest Trends Artif Intell*, *4*(4).
16. Tekale, K. M., Enjam, G. R., & Rahul, N. (2023). AI Risk Coverage: Designing New Products to Cover Liability from AI Model Failures or Biased Algorithmic Decisions. International Journal of AI, BigData, Computational and Management Studies, 4(1), 137-146. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V4I1P114
17. Abd Wahab, N. H., Hasikin, K., Lai, K. W., Xia, K., Bei, L., Huang, K., & Wu, X. (2024). Systematic review of predictive maintenance and digital twin technologies challenges, opportunities, and best practices. PeerJ Computer Science, 10, e1943.
18. Ghobadi, F., & Kang, D. (2023). Application of machine learning in water resources management: a systematic literature review. Water, 15(4), 620.

19. Wu, N., & Xie, Y. (2022). A survey of machine learning for computer architecture and systems. ACM Computing Surveys (CSUR), 55(3), 1-39.
20. Teja Thallam , N. S. (2023). Centralized Management in Multi-Account AWS Environments: A Security and Compliance Perspective. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(3), 23-31. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I3P103
21. Tekale, K. M. (2023). Cyber Insurance Evolution: Addressing Ransomware and Supply Chain Risks. International Journal of Emerging Trends in Computer Science and Information Technology, 4(3), 124-133. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I3P113
22. Hassebo, A., Obidat, M., & Ali, M. (2017, December). Four LTE uplink scheduling algorithms performance metrics: Delay, throughput, and fairness tradeoff. In 2017 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT) (pp. 300-305). IEEE.
23. Sandeep Rangineni Latha Thamma reddi Sudheer Kumar Kothuru , Venkata Surendra Kumar, Anil Kumar Vadlamudi. Analysis on Data Engineering: Solving Data preparation tasks with ChatGPT to finish Data Preparation. Journal of Emerging Technologies and Innovative Research. 2023/12. (10)12, PP 11, https://www.jetir.org/view?paper=JETIR2312580
24. Newaz, M. N., & Mollah, M. A. (2023, February). Memory usage prediction of HPC workloads using feature engineering and machine learning. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (pp. 64-74).
25. Naga Surya Teja Thallam. (2022). Cost Optimization in Large-Scale Multi-Cloud Deployments: Lessons from Real-World Applications. International Journal of Scientific research in Engineering and Management, 6(9).
26. Katya, E. (2023). Exploring feature engineering strategies for improving predictive models in data science. Research Journal of Computer Systems and Engineering, 4(2), 201-215.
27. Matlock, K., De Niz, C., Rahman, R., Ghosh, S., & Pal, R. (2018). Investigation of model stacking for drug sensitivity prediction. BMC bioinformatics, 19(Suppl 3), 71.
28. Tekale , K. M. (2023). AI-Powered Claims Processing: Reducing Cycle Times and Improving Accuracy. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 4(2), 113-123. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I2P113
29. Sehrawat, S. K. (2023). Transforming Clinical Trials: Harnessing the Power of Generative AI for Innovation and Efficiency. *Transactions on Recent Developments in Health Sectors*, *6*(6), 1-20.
30. Dimitrijevic, B., Khales, S. D., Asadi, R., & Lee, J. (2022). Short-term segment-level crash risk prediction using advanced data modeling with proactive and reactive crash data. Applied Sciences, 12(2), 856.
31. Nita, S., & Kartikawati, S. (2020, April). Analysis of the Impact Narrative Algorithm Method, Pseudocode and Flowchart Towards Students Understanding of the Programming Algorithm Courses. In IOP Conference Series: Materials Science and Engineering (Vol. 835, No. 1, p. 012044). IOP Publishing.
32. Tekale, K. M., & Enjam, G. reddy. (2023). Advanced Telematics & Connected-Car Data. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(1), 124-132. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I1P114
33. Thallam, N. S. T. (2021). Privacy-Preserving Data Analytics in the Cloud: Leveraging Homomorphic Encryption for Big Data Security. *Journal of Scientific and Engineering Research*, *8*(12), 331-337.
34. Bennett, N. (2015). Introduction to Algorithms and Pseudocode. Working paper in Project "Exploring Modelling and Computation.
35. Khan, M. A., Saqib, S., Alyas, T., Rehman, A. U., Saeed, Y., Zeb, A., & Mohamed, E. M. (2020). Effective demand forecasting model using business intelligence empowered with machine learning. IEEE access, 8, 116013-116023.
36. Mitra, A., Jain, A., Kishore, A., & Kumar, P. (2022, September). A comparative study of demand forecasting models for a multi-channel retail company: a novel hybrid machine learning approach. In Operations research forum (Vol. 3, No. 4, p. 58). Cham: Springer International Publishing.
37. Feizabadi, J. (2022). Machine learning demand forecasting and supply chain performance. International Journal of Logistics Research and Applications, 25(2), 119-142.
38. Settibathini, V. S., Kothuru, S. K., Vadlamudi, A. K., Thammreddi, L., & Rangineni, S. (2023). Strategic analysis review of data analytics with the help of artificial intelligence. International Journal of Advances in Engineering Research, 26, 1-10.
39. Garg, A. (2022). Unified Framework of Blockchain and AI for Business Intelligence in Modern Banking . *International Journal of Emerging Research in Engineering and Technology*, *3*(4), 32-42. https://doi.org/10.63282/3050-922X.IJERET-V3I4P105
40. Tekale, K. M. (2022). Claims Optimization in a High-Inflation Environment Provide Frameworks for Leveraging Automation and Predictive Analytics to Reduce Claims Leakage and Accelerate Settlements. International Journal of Emerging Research in Engineering and Technology, 3(2), 110-122. https://doi.org/10.63282/3050-922X.IJERET-V3I2P112