

AI-Driven Adaptive Distributed Caching for Enterprise Microservice Integration: A Validation Gated Learning Framework for Dynamic TTL, Admission, and Eviction Control

Abhishek Kumar Pandey

Independent Researcher, USA. ORCID: 0009-0000-0819-3298

Abstract

Enterprise microservices require distributed caches, but existing policies (one expiration policy and one eviction policy) are set ahead of time and often fail to adapt to dynamic access patterns, data mutation, and changing tenant requirements. This paper proposes VGACO, a Validation Gated Adaptive Cache Orchestration framework that dynamically learns and recommends caching policies: admission, dynamic TTL, refresh-ahead, tiering, and eviction, at a service-layer cache and a shared distributed cache layer in Kubernetes. Recommendations are validated against deterministic gates covering calibrated confidence, staleness, SLO compliance, cost, tenant fairness, security, and drift before canary promotion to production. A static fallback cache policy is always available. The controller interacts with service-owned data through cache contracts and versioned domain events. A reproducible experimental protocol compares five configurations: no cache, static, rule-based, validation-gated, and ungated. The framework positions learned caching control as a governed, reversible infrastructure for enterprise integration.

Keywords: Adaptive Caching, Machine Learning, Enterprise Systems Integration, Microservices, Dynamic Time to Live, Validation Gates

1. Introduction

In a microservice estate, an enterprise transaction might be routed from an API gateway to identity, pricing, entitlement, and catalog services, then to a relational system of record, then to an event broker, and finally to one or more partner system interfaces before a response is returned. Complexity is compounded when migrating from monolithic platforms, with microservice adopters reporting that the best practices for implementing the resulting architecture are less established than the migration process itself [1]. Each hop adds latency and introduces a new dependency. Distributed caching has become a widely adopted solution to both problems.

Caching is not a local optimization. Each service owns its data. Consumers read that data across service boundaries. Several microservices data management problems require platform-level support, as code discipline alone is insufficient to contain them [2]. Trace analysis of a large production microservices deployment shows that microservice call graphs are heavy-tailed and populated by many hot spot services [3]. This is precisely the condition under which a well-tuned cache is most effective and under which a poorly tuned policy can cause the most harm.

Most production configurations still have a fixed TTL, a single eviction policy (e.g., least recently used (LRU) or least frequently used (LFU)), and the same rules are applied overall to services and tenants (if applicable) in the same way. Such rules are typically set before deployment and rarely revised unless an incident forces a change. Evidence from workloads partially contradicts this claim. A study of in-memory key-value cache clusters at a large social platform found that more than 35% of 153 cache clusters had write ratios exceeding 30%. It also found that TTL often determines the working set rather than functioning as a minor tuning parameter [4]. A survey of learned caching and tiering to date reaches the conclusion that traditional eviction, admission, and prefetching rules are easy to use but poorly adaptable to changing workloads [5].

Learning-based policies are a viable solution: production datacenter flash cache deployments have demonstrated improved admission control over static policies through learned experience [6]. Another work systematically mapped literature on AI + microservices and the microservice life cycle and identified 16 research themes on quality attributes, AI domains, and DevOps phases [7]. An operating model is still underspecified. However, learned caching research typically evaluates models on storage or network traces

and focuses on a single caching decision in isolation. Enterprise integration practice describes cache-side access, invalidation, and ownership management patterns but does not explain how engineers should choose and enforce the policies involved. Neither stream describes how a learned controller should operate in the presence of service data ownership, domain events, tenant boundaries, service level objectives, and rollback discipline. Adaptive caching is therefore an enterprise integration problem as much as a performance tuning exercise.

The central research question is how a learned controller can adjust cache admission, TTL, refresh, and eviction policies across an integrated enterprise microservice estate while respecting latency SLOs, consistency limits, tenant isolation requirements, and cost budgets, with a deterministic recovery path. The design and experimental protocol are based on six subquestions.

RQ1. How much does learned control reduce P95 and P99 latency over static TTL caching when workloads exhibit workload changes?

RQ2. Can learned admission and eviction improve the byte hit ratio and offload the database without exceeding the approved memory budget?

RQ3. What is the accuracy of our TTL prediction with stale reads when using the threshold defined for each data class?

RQ4. Is event-driven, dependency-aware invalidation more consistent across services than expiration alone?

RQ5. What are the compute, memory, network, and cost overhead of telemetry, inference, and policy orchestration?

RQ6: How often does the validation gate reject or roll back model recommendations, and under what conditions?

First, the Validation Gated Adaptive Cache Orchestration framework is introduced, placing a learned controller on a separate control path so that inference failure reduces adaptivity without affecting availability. Second, a multi-decision learning model is introduced that covers admission, TTL, refresh-ahead, tier placement, and eviction under a shared reward function that penalizes stale reads and SLO violations more heavily than it rewards hit ratio improvements. Third, an integration model provides cache contracts, versioned domain events, and dependency-aware invalidation. Fourth, a reproducible experimental protocol with rule-based and ungated baselines tests the AI contribution empirically rather than assuming it.

This article covers application-layer and distributed data caches in cloud-native enterprise systems, excluding processor caches, content delivery networks, and purpose-built caches such as key-value stores supporting large language model inference. No measured performance results are reported; all numerical values in the paper are design parameters or pre-specified acceptance criteria for the proposed experiments. Section 2 describes related work, Section 3 describes the framework, Section 4 describes the experiment design, and Section 5 describes expected results, governance, and limitations.

2. Related Work

2.1 Learned Cache Management

One relevant line of work addresses adaptive replacement through online learning over expert policy combinations. CACHEUS extends regret-driven expert selection to more workload primitive types. The authors evaluated its performance through 17,766 simulation experiments with 329 workloads on 6 cache configurations in total [8]. The breadth of that evaluation is important for enterprise workloads, where no single heuristic is optimal across all access patterns. Deep reinforcement learning was recently applied to replacement in a cloud block storage service, and the resulting policy achieved a higher hit rate than several existing methods while also reducing request latency and backend traffic [9].

Similarly, production experiments in a datacenter flash caching deployment demonstrated a 7.7% lower total cost of ownership compared to previous static admission policies [6]. The same deployment reduced disk reads by 9.5% and flash wearout by 17.8%. These results are relevant to the proposed design, suggesting that optimizing admission independently of eviction can yield meaningful economic benefits.

These systems are typically framed in the framework of online learning, where the regret [10] is the difference between the hits of the candidate algorithm and the oracle algorithm that at all times plays the most popular items. Cost-aware formulations extend this to the case when the miss costs are also stochastic and need to be learned, with asymptotically optimal regret bounds [11]. Admission in distributed edge caches has also been modeled as a Markov decision process (MDP), where a learned agent makes the caching decision based on popularity signals [12]. Learning has also been applied to the TTL algorithm. For hierarchies of caches with random fetch latencies, the exact model becomes intractable, and a machine learning-based TTL estimation that learns optimal TTL from data has been proposed [13]. A survey work identifies eviction, admission, and prefetching as related decisions based on common feature families [5].

Two limitations are consistent across this literature. Most policies optimize a single decision, typically eviction, against a single criterion, usually hit ratio or byte hit ratio. Enterprise services require multiple decisions to be made in coordination, accounting for the risk of stale data. Evaluations are conducted on storage, network, or synthetic traces that lack service ownership boundaries, domain events, and tenant structure. Table 1 compares the VGACO against representative work along the two dimensions.

Table 1. Positioning of VGACO against representative learned caching work

Work	Decisions in scope	Setting	Integration awareness	Safety gating and fallback
[8]	Eviction	Storage traces, simulation	Not a design focus	Not a design focus
[9]	Eviction	Cloud block storage	Not a design focus	Not a design focus
[6]	Admission	Datacenter flash cache	Not a design focus	Production rollout, not formalized as gates
[11]	Caching under uncertain cost	Edge caching service	Not a design focus	Not a design focus
[12]	Admission	Distributed edge caches	Multi-node coordination	Not a design focus
[13]	TTL	Cache hierarchies with fetch delay	Not a design focus	Not a design focus
VGACO	Admission, TTL, refresh, tier, eviction	Enterprise microservices	Contracts, domain events, dependencies, tenants	Deterministic gates, canary, rollback, fallback

2.2 Caching and Data Management in Microservices

The literature of microservices identifies shared data as a structural anti-pattern. A tertiary study provides a catalog of 58 non-overlapping anti-patterns. They are grouped into five groups and are based on 203 originally identified microservice anti-patterns [14]. Unmanaged shared caches reproduce exactly the coupling these catalogs warn against: consumer services become dependent on an implied data representation owned by no single service.

A systematic review of event-driven microservices identifies three categories of consistency issues: data consistency, process consistency, and system evolution consistency [15], each of which has direct caching implications. Data consistency governs the rate of stale reads. Process consistency governs whether invalidations arrive in a form consumers can safely apply. Evolution consistency defines behavior when a schema version change occurs while entries remain cached.

Model-based predictions of performance design patterns in a microservices architecture were found to closely match actual measurements and reproduce events such as bottleneck switches [16]. These results support a staged evaluation where a trace-driven simulation can precede a controlled testbed before testing with production traffic. Using an application cache to serve requests on cache hits enables cloud-native applications to trade scalability for cost reduction without performance degradation, supporting autoscaling strategies [17]. In particular, cache size, application replicas, and inference replicas need to be controlled separately. Adding service replicas increases load on the cache but does not increase the working set size. Adding shards redistributes keys at the cost of a rebalance.

2.3 Governed Operational Machine Learning

Operational AI for cloud platforms has emerged as a distinct research area. A survey of AIOps techniques for failure management identified 100 techniques and investigated their applicability and quantitative results [18]. As the number of services and inter-service communication paths in a cloud-native application grows, anomaly detection and root cause analysis become proportionally harder [19]. MLOps literature similarly identifies the difficulty of converting models into stable production systems, with a large proportion of ML projects failing to reach production [20]. The pipeline, versioning, and monitoring practices proposed in that work form the substrate on which any production cache controller must run.

Governance of a cache controller requires three mechanisms: drift detection, calibration, and explanation. In performance-aware detectors, a drop in prediction accuracy signals a change in the monitored system [21]. A survey of more than 130 high-quality papers split the field into drift detection, calibration, and drift adaptation methods. Calibration seeks to align the classifier's confidence scores with the actual probability of a correct prediction, a precondition for using a confidence threshold as a deployment gate. The third is explanation. More recent surveys treat explainability as a necessary condition for trusted deployment of AI in critical applications [24]. The AI Risk Management Framework organizes such controls under the govern, map, measure, and

manage functions [25]. VGACO treats all three mechanisms as deterministic, version-controlled checks rather than advisory guidance.

3. Materials and Methods: The VGACO Framework

3.1 Architecture Overview

VGACO isolates the data path from the control path, routing requests through a service interface and cache, gathering telemetry asynchronously, bounding learned controller recommendations, and validating every recommendation before it reaches the request path. As the controller is never in the synchronous request path, a transient controller outage reduces adaptivity but not availability, a property that operations teams require before enabling a learned model at production scale. Figure 1 shows the arrangement, and Table 2 lists the layers.

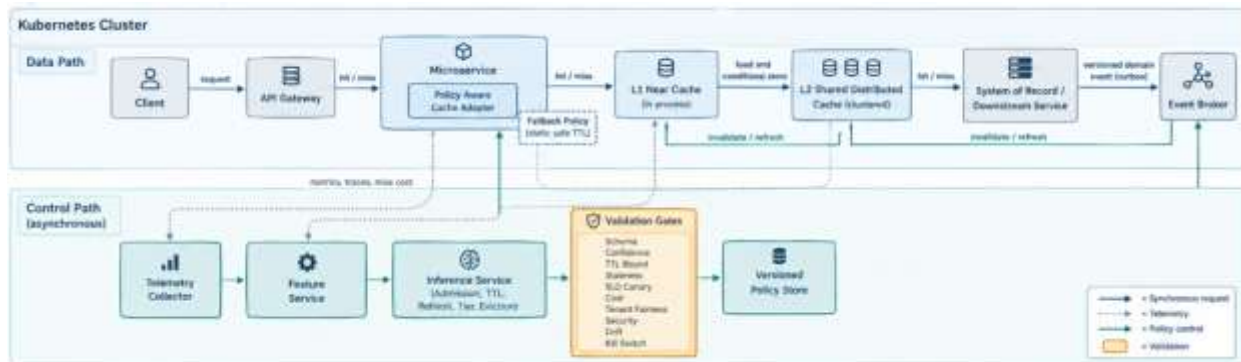


Figure 1. VGACO architecture, in which learned recommendations travel on an asynchronous control path and reach the request path only as validated, versioned policies.

Table 2. VGACO architecture layers

Layer	Representative components	Responsibility
Access and integration	API gateway, partner APIs, batch interfaces, event consumers	Carries tenant and requests context without exposing sensitive payloads to the model
Microservice execution	Spring Boot services and domain logic	Uses cache contracts and preserves system of record ownership
Cache data plane	Caffeine near cache and Redis Cluster	Provides in-process access and shared cross-pod state
Event and invalidation	Durable event broker with transactional outbox	Propagates versioned domain changes and invalidation commands
Observability	OpenTelemetry collectors and metrics store	Captures access, miss cost, latency, change rate, errors and resource use
AI decision plane	Feature service, inference service, policy store	Recommends admission, TTL, refresh, tier and eviction priority
Validation and governance	SLO, staleness, confidence, cost and security gates; audit; rollback	Prevents unsafe policies from reaching broad traffic
Kubernetes operations	Operator, Deployments, StatefulSets, autoscalers, network policy	Deploys and scales components while preserving isolation

3.1.1 Data Path

Services first check their local near cache, then the shared distributed cache, and finally call the system of record or the downstream service and store the observed miss cost if they observe a cache miss on the second lookup. It only stores the result if the current policy allows it to. Each entry contains the policy version, the data version, the identifier for the current tenant, and the time of its creation. These four fields allow the service to detect a superseded policy or an invalidation race without contacting the controller.

3.1.2 Control Path

Telemetry data is bucketed into timelines of fixed width, and a feature service extracts class, endpoint, service and tenant features. The final inference service returns a recommendation, a confidence score, expiry time and reason codes. If the validation check passes, a versioned policy is published that the services pull using either a local agent or a Kubernetes custom resource. Each service keeps the previous safe policy to allow a rollback without making a call to the control plane.

3.1.3 Decision Granularity

If keys were included in every decision, excessive overhead would be incurred and telemetry cardinality would be unbounded. Policies are instead grouped together as policy groups (e.g., tenant configuration, authorization metadata) alongside catalog objects, dashboard summaries, and external reference data. For high-value keys, this can be done per individual; for large populations, a group policy can balance precision and cost.

3.2 Feature Model

The controller defines only metadata, no business payloads. Table 3 shows the different feature families. Demand features estimate near-term reuse. Change features estimate stale data risk. Cost features assess the penalty of a miss, and object and system features constrain object, sensitivity, and delivery capacities. Business and dependency features deal with business concerns and invalidate data for a specified scope. Model context features control the lifecycle of models. Each tenant-level feature is normalized to prevent higher-volume tenants from dominating training.

Table 3. Feature families used by the learned controller

Feature family	Examples	Purpose
Demand	Access count, inter-arrival time, recency, unique callers, burst score	Estimates near-term reuse
Change behavior	Update count, change interval, invalidation frequency, schema version	Estimates stale data risk
Retrieval cost	Database time, partner API time, CPU cost, bytes transferred	Measures miss penalty
Object properties	Serialized size, compression ratio, sensitivity, consistency class	Constrains storage and safety
System state	Memory pressure, eviction rate, CPU, network, pod count	Represents current capacity
Business context	Tenant tier, endpoint criticality, service SLO, transaction type	Aligns policy with enterprise priority
Dependency context	Upstream owner, downstream consumers, event lag, dependency depth	Supports integration aware invalidation
Model context	Policy age, confidence, drift score, last rollback reason	Supports safe lifecycle control

3.3 Learned Decision Model

The controller produces six outputs (Table 4). The motivation for low inference latency, tabular features, and explainability is that the operator must read why a policy changed to trust that change. Reinforcement learning is also not considered until supervised and bandit baselines have been proven insufficient because a simpler model is easier to explain, debug, and roll back.

Table 4. Controller outputs

Output	Form	Use
Admission probability	0 to 1	Store the item only when the expected reuse value exceeds the memory and staleness cost
Predicted TTL	Seconds	Bounded by domain minimum and maximum values
Refresh probability	0 to 1	Schedule refresh ahead for items likely to remain hot
Eviction priority	Rank score	Select lower-value objects when memory is constrained
Cache tier	None, local, distributed, or both	Choose placement from sharing and latency needs
Confidence and reason codes	Score plus feature attributions	Feed gates, audit records, and operator review

3.3.1 Admission

The admission problem is a binary classification problem. Object i is positive if it would be used again before its useful lifetime expired and before it was evicted. These labels can be computed offline from the access and update traces. This is satisfied if the expected probability of future reuse multiplied by the expected saved miss cost exceeds the memory cost plus staleness cost. Gradient boosted trees, which are preferred, are good at learning from tabular operational data, can predict quickly within a limited time budget, and expose feature attributions.

3.3.2 Dynamic TTL

TTL prediction uses regression constrained by business rules, as shown in (1).

$$TTL_i = \text{clip}(TTL_{\min,c}, TTL_{\max,c}, g(F_i, R_i, U_i, C_i, S_i, M, T_i)) \quad (1)$$

Here, F is access frequency, R is recency, U is update rate, C is miss or computation cost, S is stale data sensitivity, M is memory pressure, T is tenant or business priority, and c is the consistency class of the data. The clip function restricts the output to the considered class's certified set. The offline training target is the safe useful period of the cache, whose length is the minimum of the time until the next update and the time until the last reuse of the object. Due to the double cost of incorrect TTLs (too short means missing a read, too long means reading stale data), the model is trained as a quantile regressor with a loss function biased toward underestimating the TTL.

3.3.3 Eviction Utility and Policy Selection

Eviction ranks entries by the utility in (2), and the lowest utility entries that the policy approves are removed first.

$$Utility_i = \alpha \cdot Reuse_i + \beta \cdot MissCost_i + \gamma \cdot Priority_i - \delta \cdot Size_i - \varepsilon \cdot StalenessRisk_i \quad (2)$$

Weights are tuned on the validation set before being saved for each model version. If supervised ranking does not determine which option to choose, the controller can use a contextual bandit to choose from a small set of policy variants within a group. For example, with a known group feature vector and arms consisting of base eviction policy and TTL multiplier, regret against the best fixed variant indicates how quickly the controller can learn [10].

3.3.4 Reward and Safety Weighting

Learning and offline evaluation both use the reward in (3).

$$\text{Reward} = \text{SavedMissCost} - \text{MemoryCost} - \text{StaleReadPenalty} - \text{InferenceCost} - \text{SLOViolationPenalty} \quad (3)$$

Penalty weights can be configured such that one stale read in a strict consistency class outweighs many further hits, preventing the controller from maximizing hit ratio by trading correctness for efficiency. Other classes, such as payment status, account balances, and clinical decision data, can be marked as non-cacheable or governed by strict rules that take precedence over the model.

3.3.5 Explanation and Reason Codes

For every recommendation, top-feature attributions (computed with the tree explanation algorithm), the inferred reason codes (e.g., HIGHREUSE, VOLATILESOURCE, MEMORY_PRESSURE), and the policy version are all recorded in the audit log. When looking at the audit log for a recommendation, an operator can see things such as the TTL being shortened due to an increase in source update rate. This means a model decision was transformed into an operational fact, which can then be checked.

3.4 Enterprise Integration Model

3.4.1 Data Ownership and Cache Contracts

Each cached data class has a single definitive ownership and publishes a cache contract; the consumer service can cache a derived representation but does not change the ownership rule. This avoids the situation where multiple services create cache semantics for the same business object, which is the failure mode that anti-pattern catalogs describe in general terms. The fields of the contract are shown in Table 5.

Table 5. Cache contract fields

Contract field	Example	Purpose
Data owner	tenant configuration service	Identifies the system of record
Key template	prod:{tenant}:config:{id}:v3	Prevents collision and supports versioning
Consistency class	bounded staleness	Selects validation rules
Maximum staleness	Class-specific ceiling in seconds	Caps the model predicted TTL
Invalidation event	TenantConfigurationChanged.v3	Coordinates consumers
Sensitive fields	Credentials excluded	Prevents unsafe serialization
Fallback	System of record with rate limit	Defines behavior during cache outage
Policy owner	Platform performance team	Assigns accountability

3.4.2 Event-Driven Invalidation

TTL alone cannot guarantee timely consistency, so the owner publishes the versioned domain event with its changes as a transactional outbox entry after committing its current transaction. In the comparison of event and cached versions, the consumer either evicts or updates the relevant key and also populates the event lag as a feature. Notifications from the cache store are useful for monitoring but not a durable business signal. In a clustered cache, every node sends a notification; thus, business invalidation is based on domain events.

3.4.3 Dependency-Aware Refresh

If the composite API caches a response from several services, then it records the identifiers of the source classes in its policy record. In response to the source event, the invalidation service follows the dependency graph and removes all affected composites. The graph's depth is limited to avoid invalidation storms, and batch invalidation, key tags, and versioned namespaces keep the operational costs predictable.

3.4.4 External Systems

Partner and vendor APIs have variable latency and strict rate limits. They generally produce good cache candidates but have different freshness guarantees. The framework tracks vendor TTL limits, legal and regulatory requirements, retry cost, and error handling for each integration and allows a stale-if-error policy. This policy may be suitable for low-risk reference data but not for responding to transactions. This makes the integration policy explicit, not tacit.

3.4.5 Modernization of Legacy Services

Legacy applications connect to the system through an API gateway, sidecar, or shared caching adapter. The framework is adopted by first adding observability, then read-only cache aside access, event-driven invalidation, and finally adaptive control. Implementing this staged path should improve performance without requiring a rewrite and provide evidence for microservices decomposition if needed.

3.5 Validation-Gated Control

Even after being recommended, a model is only an advisory recommendation until it passes through every gate. Gates are deterministic, version controlled, run before canary deployment, and run continuously at runtime. If the gate fails, the last-known-good policy is restored. The gates are summarized in Table 6, and the policy lifecycle is shown in Figure 2.

Table 6. Validation gates and failure actions

Gate	Check	Failure action
Schema and feature	Required features present, typed, and within the trained range	Reject and use deterministic default
Confidence	Calibrated confidence exceeds the threshold for the data class	Retain current policy
TTL boundary	Predicted TTL inside domain and compliance limits	Clip or reject
Staleness	Projected stale read rate below the contract limit	Shorten TTL or disable caching
SLO	Canary P95, P99 and error rate within budget	Automatic rollback
Cost	Memory and inference cost within the approved savings ratio	Do not promote
Tenant fairness	No tenant consumes a disproportionate share of capacity	Apply quota or separate policy
Security	Sensitive or prohibited data excluded; key and payload rules pass	Block deployment and alert
Drift	Feature and outcome drift within threshold	Freeze model and retrain
Kill switch	Operator or incident automation disables AI control	Return to static safe policy

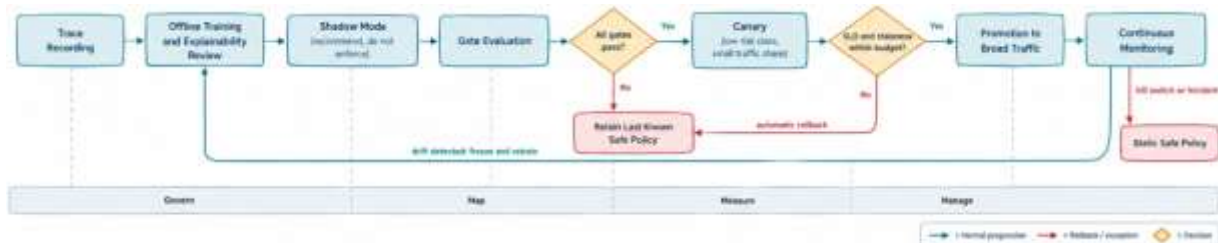


Figure 2. Validation gated policy lifecycle showing shadow evaluation, gate checks, canary promotion, drift-driven retraining, and deterministic rollback to the last known safe policy.

Canary promotion progresses from a nonproduction environment to a low-risk service or tenant to a small percentage of live traffic. Promotion takes place after a minimum observation window and a sufficient number of cache events have occurred. It is suspended during incidents, major deployments, or during windows of insufficient telemetry data. Fallback is built-in rather than an exception path. Each cache class has a TTL, a valid eviction policy, and a way to communicate with the system of record, and the inference client has a strict timeout. If the model is not available, it keeps using its last accepted policy and never waits for retraining or model recovery.

The gates map onto the four functions of the AI Risk Management Framework [25]. The policy ownership and kill switch gates implement governance. Contracts and sensitivity classification implement the map function. Calibration, drift, and SLO canary checks implement the measure function. Rollback and policy retirement implement the manage function. This mapping gives audit and risk teams a vocabulary they already know, so the approval path for a learned component is shortened.

3.6 Cloud Native Reference Implementation

The reference implementation runs on Java 17+ and uses Spring Boot and its cache abstraction for business services, Caffeine as an in-process cache, Redis Cluster as a shared cache, a durable event broker carrying invalidation announcements, and OpenTelemetry to collect traces and metrics. The inference service can be run on a JVM or in Python. An optional exchange format is ONNX; every lookup is wrapped in a policy-aware cache adapter. If a decision is available when the regular get or load operation occurs, it will be used. If not, or if the decision was rejected, the key class policy is used instead, which performs a get or load with the accepted TTL and tier. Business logic is separate from model logic. The adapter records the policy version and outcome and enforces tenant key prefixes, serialization rules, and stale records limits.

Policies are versioned custom resources and are signed as part of normal deploys. For example, a tenant config policy locally constrains TTLs to between 15 and 300 seconds, stale reads below 0.001, inference timeouts below 3 ms, a fallback TTL of 60 seconds, LFU eviction on all keys, and a canary that serves 5% of the traffic over a minimum window of 30 minutes. These are design and not measured goals, as the shared cache has its own memory limits and an eviction policy to fall back on if needed. The controller cannot assume that every model-ranked eviction actually happens when memory pressure occurs. The near cache TTL should never be higher than the distributed TTL.

The inference service, feature service, validator, and operator run as separate Kubernetes Deployments (to allow for independent scaling), and the shared cache runs as an approved managed service or with a carefully controlled StatefulSet. Anti-affinity rules, disruption budgets, readiness probes, resource requests and limits, and zone placement allow the services to be available, while network policies and workload identity ensure only authorized services access the cache.

4. Experimental Design

4.1 Testbed

The testbed is deployed to Kubernetes and consists of an API gateway, tenant configuration service, catalog service, authorization service, reporting service, and an integration adapter for a partner API simulation. The backend includes a PostgreSQL system of record, Redis Cluster, Caffeine, an event broker, OpenTelemetry collectors, and the VGACO control plane. Using K6, Gatling, or JMeter as a load generator, all tests are performed against the same application build and data set. Both features and weights are tuned using a trace-driven simulation before the testbed is run using the staged approach described in Section 2.2.

4.2 Comparison Configurations

To analyze the contribution of each design, five configurations are defined in Table 7. C2, the rule-based configuration, is essential because it separates the value of learning from the value of adaptation in general. The ungated configuration C4 is only used under test conditions, where the value of the validation gates can be measured.

Table 7. Comparison configurations

Configuration	Description	Purpose
C0: No cache	All reads reach the system of record or downstream service	Establishes source system cost and latency
C1: Static cache	Cache aside with fixed TTL and default store eviction	Represents common enterprise practice
C2: Rule-based adaptive	Handwritten thresholds adjust TTL and refresh	Separates AI value from general adaptation
C3: VGACO	Learned admission, TTL, refresh, tier, and eviction with gates	Tests the proposed contribution
C4: VGACO without gates	Learned control with gates disabled, isolated tests only	Measures the value of validation controls

4.3 Workload Profiles

The eight workload profiles in Table 8 represent conditions under which static policies are expected to fail. The failure injection profile is run against each configuration to demonstrate rollback and fallback behavior.

Table 8. Workload profiles

Profile	Behavior	Evaluation focus
Stable read heavy	Predictable Zipf-like popularity and low update rate	Steady-state benefit
Burst	Rapid traffic increase for a subset of keys and tenants	Adaptation speed and stampede control
Churn	Hot key set changes frequently	Eviction and admission quality
Volatile data	Update rate rises while reads remain high	TTL and stale read control
Large object mix	Small and large objects share one memory budget	Byte aware utility
Multi-tenant skew	One tenant becomes disproportionately active	Quota and fairness gates
Partner API degradation	Partner latency and error rate increase	Stale if error and retrieval cost response
Failure injection	Cache node loss, inference timeout, event lag, telemetry loss	Fallback and rollback

4.4 Scale, Repetition, and Failure Injection

A test run is performed for each configuration with 100, 500, 1000, and 5000 virtual users or the same rate for that configuration on the environment and k fractions of the working set as the cache size. Each run is composed of a warm-up phase, steady-state phase, workload change, and recovery phase. At least 30 of each configuration will be tested for inferential comparison as the budget allows, with control plane failure injections performed first. These can include the inference service being unavailable or having a longer return time than its timeout, a missing, delayed, or out-of-distribution feature stream, and infrastructure issues, such as a cache shard failing over to a backup shard or a lagging event broker. Policy faults include recommending a TTL higher than the maximum, creating a canary that exceeds its P99 or stale read gate, and rollback failing under peak traffic. A legacy service reading an old contract version tests backward compatibility.

4.5 Evaluation Measures

System measures are reported together with the model measures, since neither is sufficient alone. Table 9 shows these measures and their questions.

Table 9. System and model evaluation measures

Category	Measure	Question answered
Latency	Mean, median, P95, P99, maximum	How do normal and tail response times change?
Throughput	Requests and successful transactions per second	Does service capacity change?
Cache effectiveness	Object hit ratio, byte hit ratio, miss rate	How much reuse is achieved?
Backend offload	Database queries, CPU time, partner calls	How much work is avoided?
Correctness	Stale read rate, invalidation lag, version mismatch	Is consistency preserved?
Resources and cost	Cache memory, application and inference CPU, network, cost units	What does the controller cost?
Reliability	Errors, timeouts, fallback use, recovery time	Is resilience preserved?
Fairness	Capacity share and hit ratio by tenant tier	Is any tenant starved?
Admission quality	Precision, recall, F1, value weighted precision	Were admitted objects reused?
TTL quality	Mean absolute error and quantile loss against the safe useful period	How close was the predicted TTL?
Eviction quality	Regret against the best tested baseline, byte hit difference	Were higher value objects retained?
Calibration	Expected calibration error, reliability diagrams	Did confidence match observed correctness?
Drift and governance	Drift alarms, gate rejections, rollbacks, overrides	How often did safety controls intervene?

4.6 Statistical Analysis Plan

Latency distributions are skewed; medians and percentiles are reported with bootstrap confidence intervals. For each run over the same trace, the Wilcoxon signed-rank test is applied and Cliff's delta is reported as the effect size measure. The family-wise error rate is controlled using the Holm procedure across the six research questions. For calibration, reliability diagrams and expected calibration error are used following classifier

calibration practice [23]. Drift is monitored with performance-aware detectors [21]. Time series plots show the adaptation and rollback behavior for each run, as final averages do not convey important transitions.

4.7 Acceptance Criteria

The conditions under which each claim may be applied are given in Table 10 and are provided in advance of the experiment so that no claim may be made afterwards based on whichever metric has improved.

Table 10. Acceptance criteria

Area	Criterion	Claim enabled
Performance	C3 improves P95 or P99 over C1 without a higher error rate	Performance claim
Correctness	Stale read rate stays within the cache contract limit	Safe promotion
Economic value	Backend and infrastructure savings exceed controller cost by an approved ratio	Business value
Inference safety	P99 inference latency within budget and fallback succeeds	Operational viability
Resilience	Rollback completes within the defined recovery objective	Production readiness
AI contribution	C3 outperforms C2 on at least one meaningful metric without violating a gate	Value beyond rule-based adaptation

5. Results and Discussion

As the Section 4 experiments have not yet been conducted, this section reports design analysis and predictions together with conditions for falsification, but not statistics for VGACO performance, expected cost savings, or prediction accuracy.

5.1 Where Learned Control Should and Should Not Help

Learning control is expected to outperform in bursty, high-churn, and volatile data access patterns where a static policy tuned for one regime does not work well in other regimes. For the stable read-heavy profile, VGACO performs as well as a well-tuned static TTL. Such an outcome would still be useful in that it would identify the classes of cache for which AI control is costly but not helpful, and since the design of the framework makes it easy to revert any policy group back to static control; this would still be an option.

5.2 Why Hit Ratio Alone Misleads

A higher hit ratio does not necessarily mean a better controller, as a controller may be increasing the number of hits by caching objects that are large or stale. This interpretation requires considering tail latency, backend offload, stale reads, memory, inference overhead, tenant fairness, and cost. A result is acceptable if it improves the desired goal without breaking the internal gate. The effect of the reward in (3) is what is seen here.

5.3 The Rule-Based Baseline and the Gate Ablation

If VGACO does not perform at least as well as the rule-based thresholds of C2, the simpler algorithm is superior, and the study must report that conclusion. The design also has to test the contribution of AI and not just endorse it. If the above proves true, that is, if C4 has a higher raw hit ratio than C3 at the cost of stale reads, SLO violations, or the like, then enterprise AI will require governance and rollback, rather than being subject to unlimited optimization. This analysis would also quantify the cost of validation gates in terms of forgone hit ratio.

5.4 Security, Privacy, and Tenant Governance

For a metadata model, access count, object size, tenant tier, change rate, and miss cost will generally suffice. It may be irrelevant to store user identifiers, in which case they can simply be hashed or aggregated. Credentials, tokens, payment information, and other regulated data are not stored unless authorized use cases require it. Keys contain an environment and tenant namespace. Workload identity supports both service-to-policy and service-to-cache access. Tenant quotas prevent a noisy tenant from consuming shared memory. Model artifacts, feature definitions, and policies are versioned and cryptographically signed. Training lineage is logged. Unknown model versions are not accepted. A compromised model service cannot circumvent the validator or write directly to the cache data plane. A cache infrastructure failure must never weaken an authorization, because the system of record is authoritative.

5.5 Adoption Path

The eight phases of adoption each have an exit condition. The first phase instruments services and identifies the cache classes and establishes no cache and static cache baselines. The second creates contracts, key standards, tenant quotas, and event schemas, while the third provides rule-based adaptation together with the validation gates so that the safety controls are proved before any model comes into play. Stage four is an offline admission, TTL and eviction model, built from the collected traces, and subject to explainability review. In stage five, the models are run, shadowing the production traffic to provide admission, TTL, and eviction recommendations that are compared with production decisions that were not enforced. Stage six is used for all

other non-medium-risk classes and bounded canary traffic. The seventh is more services, tenants, and integration dependencies, and the eighth is drift monitoring, retraining, cost reviewing, and retiring less valuable policies.

5.6 Limitations and Threats to Validity

The biggest risk is workload representativeness. Synthetic tests cannot model real tenant workloads, event delays, or failures in the downstream. Anonymized production traces should be replayed whenever policy allows, and performance of the models should be evaluated on the extracted features. When telemetry is dropped or delivered late, the model may take bad decisions. Feature gates and fallbacks are not bulletproof. Choosing a cache technology is non-trivial due to differences in eviction policies, behavior in a cluster, and many other factors. The initial experiment is therefore limited to the specified cache technology; results may not generalize across the broader microservice portfolio without further validation. Repeating this experiment on healthcare, finance, and commerce workloads would improve external validity, as the limits on sensitivity and consistency may be different and the costs are dependent on the cloud pricing and the shape of the deployment. The resource units and formulae should be provided alongside the currency costs. Additionally, some resources provided by the framework may not be justified for the steady-state service. Therefore, static and rule-based alternatives remain first-class options, and the expected outcome may be to only apply learning to a few high-variance cache classes.

6. Conclusion

While static cache policies operate in static workloads, enterprise microservice estates do not. It is unclear how a learned controller could adapt admission-control, TTL, refresh, and eviction policies of interconnected services to meet (i) latency requirements, (ii) consistency requirements, (iii) tenant isolation requirements, (iv) cost budgets, and (v) deterministic recoverability. The answer here is structural. VGACO runs its model off the request path, restricts its recommendations to policy groups, passes all recommendations through version-controlled gates before canary, and can always count the static safe policy as one step away from a correct recommendation. A learned decision model within this envelope nets admission as classification, TTL as short-lifetime-favoring predictive loss, eviction as size, staleness risk, and reuse utility, and all decisions as correctness-favoring reward over hit ratio. Cache contracts and versioned domain events bind learned model decisions to the ownership boundaries that enterprise integration already recognizes, allowing invalidation to be driven by business events rather than expiry timers or individual cache clocks.

This acts as a testable operating model for learned caching in an enterprise microservices setting, although it makes no claims of actually measured improvements; the experiment protocol is constructed to provide a strong basis for rejecting the hypothesis. This only tests the benefit of learning against adaptation; the ungated version also tests its cost and harmful effects. The thresholds and acceptance are determined prior to execution. Assuming learned control is needed only for a small fraction of volatile and high-traffic classes of the cache, the rest of the system can revert to the static policy specified by their contracts. The simplest way to validate this is to run the protocol with replayed enterprise traces from multiple domains.

Declarations

Conflict of Interest: The author declares no conflict of interest.

Statement of Human and Animal Rights: This article does not contain any studies with human or animal subjects performed by the author.

Informed Consent: Not applicable, as no human participants were involved in this study.

AI Tool Disclosure: Generative AI tools were used to assist with drafting and language editing under the author's direction. The author reviewed, verified, and revised all content, including every reference, and takes full responsibility for the manuscript.

References

- [1] Victor Velepucha and Pamela Flores, "A Survey on Microservices Architecture: Principles, Patterns, and Migration Challenges," *IEEE Access*, vol. 11, pp. 88339–88358, Aug. 2023. Available: <https://doi.org/10.1109/ACCESS.2023.3305687>
- [2] Rodrigo Nunes Laigner et al., "Data Management in Microservices: State of the Practice, Challenges, and Research Directions," *Proceedings of the VLDB Endowment*, vol. 14, no. 13, pp. 3348–3361, 2021. Available: <https://doi.org/10.14778/3484224.3484232>

- [3] Shutian Luo et al., "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis," *Proc. ACM Symposium on Cloud Computing (SoCC '21)*, pp. 412–426, Nov. 2021. Available: <https://doi.org/10.1145/3472883.3487003>
- [4] Juncheng Yang et al., "A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter," *ACM Transactions on Storage*, vol. 17, no. 3, Art. 17, Aug. 2021. Available: <https://doi.org/10.1145/3468521>
- [5] George Savva et al., "Machine Learning-Based Caching and Tiering in Modern Data Storage Systems: A Survey," *ACM Computing Surveys*, vol. 58, no. 11, pp. 1–36, Mar. 2026. Available: <https://doi.org/10.1145/3803857>
- [6] Tzu-Wei Yang et al., "CacheSack: Theory and Experience of Google's Admission Optimization for Datacenter Flash Caches," *ACM Transactions on Storage*, vol. 19, no. 2, Art. 13, Mar. 2023. Available: <https://doi.org/10.1145/3582014>
- [7] Sergio Moreschini et al., "AI Techniques in the Microservices Life Cycle: A Systematic Mapping Study," *Computing*, vol. 107, no. 4, Art. 100, Apr. 2025. Available: <https://doi.org/10.1007/s00607-025-01432-z>
- [8] Liana V. Rodriguez et al., "Learning Cache Replacement with CACHEUS," *Proc. 19th USENIX Conference on File and Storage Technologies (FAST '21)*, pp. 341–354, Feb. 2021. Available: <https://www.usenix.org/conference/fast21/presentation/rodriguez>
- [9] Yang Zhou et al., "An Efficient Deep Reinforcement Learning-Based Automatic Cache Replacement Policy in Cloud Block Storage Systems," *IEEE Transactions on Computers*, vol. 73, no. 1, pp. 164–177, Jan. 2024. Available: <https://doi.org/10.1109/TC.2023.3325625>
- [10] Archana Bura et al., "Learning to Cache and Caching to Learn: Regret Analysis of Caching Algorithms," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 18–31, Feb. 2022. Available: <https://doi.org/10.1109/TNET.2021.3105880>
- [11] Guocong Quan et al., "Minimizing Edge Caching Service Costs Through Regret-Optimal Online Learning," *IEEE/ACM Transactions on Networking*, vol. 32, no. 5, pp. 4349–4364, 2024. Available: <https://doi.org/10.1109/TNET.2024.3420758>
- [12] Ting Wang et al., "Towards Intelligent Adaptive Edge Caching Using Deep Reinforcement Learning," *IEEE Transactions on Mobile Computing*, vol. 23, no. 10, pp. 9289–9303, Oct. 2024. Available: <https://doi.org/10.1109/TMC.2024.3361083>
- [13] Karim S. Elsayed et al., "Optimizing TTL cache hierarchies under random delays: Direct methods and learning on graph transformations," *Computer Networks*, vol. 280, Art. 112180, May 2026. Available: <https://doi.org/10.1016/j.comnet.2026.112180>
- [14] Tomas Cerny et al., "Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study," *Journal of Systems and Software*, vol. 206, Art. 111829, Dec. 2023. Available: <https://doi.org/10.1016/j.jss.2023.111829>
- [15] Joaquim Luz Bação and Sérgio Guerreiro, "Consistency challenges in event-driven microservices: a literature review on data, process, and system evolution," *Computing*, vol. 108, no. 7, Art. 110, Jun. 2026. Available: <https://doi.org/10.1007/s00607-026-01701-5>
- [16] Willem Meijer et al., "Experimental evaluation of architectural software performance design patterns in microservices," *Journal of Systems and Software*, vol. 218, Art. 112183, Dec. 2024. Available: <https://doi.org/10.1016/j.jss.2024.112183>
- [17] Byeonghui Jeong and Young-Sik Jeong, "Autoscaling techniques in cloud-native computing: A comprehensive survey," *Computer Science Review*, vol. 58, Art. 100791, Nov. 2025. Available: <https://doi.org/10.1016/j.cosrev.2025.100791>
- [18] Paolo Notaro et al., "A Survey of AIOps Methods for Failure Management," *ACM Transactions on Intelligent Systems and Technology*, vol. 12, no. 6, pp. 1–45, Nov. 2021. Available: <https://doi.org/10.1145/3483424>
- [19] Jacopo Soldani and Antonio Brogi, "Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey," *ACM Computing Surveys*, vol. 54, no. 3, pp. 1–39, 2022. Available: <https://doi.org/10.1145/3501297>
- [20] Dominik Kreuzberger et al., "Machine Learning Operations (MLOps): Overview, Definition, and Architecture," *IEEE Access*, vol. 11, pp. 31866–31879, 2023. Available: <https://doi.org/10.1109/ACCESS.2023.3262138>
- [21] Firas Bayram et al., "From concept drift to model degradation: An overview on performance-aware drift detectors," *Knowledge-Based Systems*, vol. 245, Art. 108632, Jun. 2022. Available: <https://doi.org/10.1016/j.knosys.2022.108632>

- [22] Jie Lu et al., "Learning under Concept Drift: A Review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, Dec. 2019. Available: <https://doi.org/10.1109/TKDE.2018.2876857>
- [23] Telmo Silva Filho et al., "Classifier calibration: a survey on how to assess and improve predicted class probabilities," *Machine Learning*, vol. 112, no. 9, pp. 3211–3260, 2023. Available: <https://doi.org/10.1007/s10994-023-06336-7>
- [24] Rudresh Dwivedi et al., "Explainable AI (XAI): Core Ideas, Techniques, and Solutions," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–33, 2023. Available: <https://doi.org/10.1145/3561048>
- [25] Elham Tabassi, "Artificial Intelligence Risk Management Framework (AI RMF 1.0)," *National Institute of Standards and Technology, NIST AI 100-1*, Jan. 2023. Available: <https://doi.org/10.6028/NIST.AI.100-1>