International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(21s) 2026 pp. 1902-1916
https://doi.org/10.70917/ijcisim-2026-5967
Article

Intelligent API Health Monitoring and Auto-Healing System with Dynamic Scaling for Cloud-Native Microservices

Anil Kumar Chitiprolu

Independent Researcher, USA
ORCID: 0009-0009-1174-4295

Abstract: APIs support communication between services in systems that utilize a microservices architecture, so they must always be operational for the systems to function properly. Traditional API monitoring solutions can produce alerts to notify administrators of API issues, but they rely on human intervention to restore API operation, leading to gradual downtime. This paper describes an Intelligent API Health Monitoring and Auto-Healing System (IAHMAS), which is a cloud-native implementation of an integrated system capable of real-time health monitoring, automated failure detection, self-healing and automated scaling. To implement IAHMAS, you need to perform periodic heartbeat checks on monitored API endpoints, use a customizable failure threshold detection mechanism to trigger automated failover logic and service instance restarts, and apply Kubernetes Horizontal Pod Autoscaler (HPA)-based compute resource scaling when service degradation occurs due to load. The three quantitative metrics defined to measure the performance of the system are the System Availability Rate (SAR), defined as the fraction of the total observation time during which all the APIs that are subject to observation remain available; the Mean Time to Recovery (MTTR), defined as the average recovery time from the API's unavailability to its recovery; and the Auto-Healing Efficiency (AHE), defined as the fraction of unsuccessful incidents that lead to auto-healing. The system was tested on five enterprise API services over 30 days, including 50 failure scenarios. It achieved a SAR of 99.99%, an MTTR of 9.9 seconds, and an AHE of 96.0%. The improvement is significant compared to the baseline of a SAR of 99.54% and an MTTR of 28.4 minutes when failure recovery is performed manually. Over 99% mean recovery time improvement is achieved over existing approaches, as well as high-availability service levels consistent with enterprise-grade SLA requirements, and we show that our approach is both practical and deployable for automated API reliability management in cloud-native systems.

Keywords: API Health Monitoring, Auto-Healing, Auto-Scaling, Kubernetes, Prometheus, Grafana, Fault Tolerance, Microservices, Cloud-Native, Site Reliability Engineering

1. Introduction

APIs are the connective tissue of modern enterprise software. Each application function in a microservices architecture can be an independently deployable service that communicates via APIs with other microservices. Therefore, the overall reliability of the system is an emergent property of the individual API endpoints [4, 5], and the failure of one API can trigger the degradation of the whole system and all the business processes dependent on it, far beyond the one API that failed. According to Richardson, communication failures between services are one of the most important reliability concerns in microservices architecture: every service call to another service represents a potential reliability failure mode that must be designed for [4].

One historic solution to help ensure API reliability has been to use monitoring tools like Prometheus and Grafana that collect metrics and can sound alerts when the system begins to deviate from normal behavior [3, 8, 13]. Although these solutions can provide rich information about the system's performance, they require humans to take action. Contrarily, using manual intervention introduces a structural delay in recovery that is not well-suited to the


Copyright: © 2026 by the authors

high availability demands of contemporary enterprise applications [7, 15]. Beyer et al. identify Mean Time to Recovery (MTTR) as an important dimension for gauging operational maturity of production systems; organizations with a matured Site Reliability Engineering (SRE) practice strive for low MTTR values measured in seconds rather than minutes [7]. Since recovery steps are usually manual and require acknowledgement, diagnosis, and remediation, MTTR numbers for non-trivial API failures are typically between 20 and 60 minutes [12, 15].

Container orchestration platforms such as Kubernetes expose service self-healing primitives such as liveness probes, which automatically restart unhealthy containers, or Horizontal Pod Autoscaler (HPA) policies, which also scale the number of replicas in a container deployment based on the demand for CPU or memory utilization [2]. However, Kubernetes' native health management operates at the container level and does not natively integrate with API health semantics at the application layer, monitoring pipelines, or intelligent failover logic [9, 11]. A gap exists between what Kubernetes can detect, container-level crashes and resource saturation, and what constitutes an API-level failure, including slow response times, elevated error rates, and partial degradation that does not cause container termination.

This paper proposes the Intelligent API Health Monitoring and Auto-Healing System (IAHMAS), which closes this gap by integrating application-layer health surveillance with automated remediation and dynamic scaling in a unified cloud-native framework. IAHMAS performs periodic heartbeat-based health checks against configured API endpoints, applies multi-tier failure detection using configurable response time thresholds and error rate windows, triggers automated failover to backup API instances when primary endpoints fail, orchestrates Kubernetes-native service restarts and pod scaling for persistent failures, and maintains comprehensive audit logs for post-incident analysis. The system is designed to achieve MTTR values consistent with automated recovery, targeting sub-15-second recovery for the majority of common failure classes while maintaining zero-touch operation that eliminates the manual recovery delay entirely.

2. Related Work

A. API Monitoring and Observability

Prometheus, developed by SoundCloud and now a Linux Foundation project, is the de facto standard for metrics collection in cloud-native environments [3]. It employs a pull-based scraping model in which monitored services expose metrics endpoints, and Prometheus collects these metrics at configurable intervals. Prometheus also supports alerting rules that trigger alerts if given conditions are true for any of the collected metrics. Prometheus can be integrated with notification services. Grafana provides real-time dashboards based on Prometheus metrics [13]. Jani describes how uniform, integrated deployments of Prometheus and Grafana enable observability through complete metrics coverage of service latencies, errors, and resource consumption, making the tool stack a scalable observability proposition for production-grade microservices [19]. Aich and Sengupta summarize deployment patterns for using Prometheus and Grafana for observability and debugging in cloud-native modernization projects, covering metric instrumentation, PromQL query design, and alert rule definition [8]. Hämäläinen et al. provide a thorough technical overview on utilizing Prometheus and Grafana in a Kubernetes cluster, from metric exposition to Prometheus scraping to Grafana visualization and alerting [20].

Mehdi et al. reviewed Grafana as a real-time monitoring and visualization system. They found that a well-designed Grafana dashboard can reduce the mean time to acknowledgment of an alert, as well as provide visual trends that allow an operator to determine when an anomalous trend began [13]. Sharma discussed multi-cloud Kubernetes observability with Istio, Prometheus and Grafana and showed that observability was consistent across multiple cloud vendors without any instrumentation changes [10]. Abirami et al. studied monitoring and alerting for horizontal auto-scaling pods in Kubernetes using Prometheus, showing that the Prometheus-based HPA metric feed reduces scaling latency compared to the native Kubernetes metrics server [9].

B. Auto-Healing and Self-Recovery

According to a systematic review of resilient microservices recovery patterns by Mohammad, there are five main families of recovery patterns: retry, circuit breaker, fallback, bulkhead and timeout. Of these, the automated implementation of recovery patterns has the highest potential impact on reducing microservices outages' mean time to repair (MTTR). Using pattern combinations achieved MTTR reductions of 70% to 90% vs. the manual baseline [15]. Fowler's circuit breaker pattern provides a vehicle for automated failure isolation: after the error rate of a monitored service exceeds a configured threshold, the circuit breaker opens and traffic is routed to a fallback service until the underlying service has recovered [6]. Miranda et al. reviewed fault-tolerant microservices design patterns such as retry, bulkhead, and fallback mechanisms that can be applied in orchestrated service architectures to create resilient systems [12].

When Oyeniran et al. reviewed design patterns for scalability in cloud-native microservices architectures, they found that traffic routing based on health checks and replica-level redundancy provided the best availability during service failure [11]. In a systematic review, Mohammad determined that automatic recovery orchestration is the most prominent gap in microservices resilience frameworks because production systems use manual escalation paths extensively, weakening the availability advantages of distributed architectures [15]. Wadhwa has studied DevOps in enterprise systems engineering with event-driven microservices, showing that deploying automated remediation pipelines integrated with monitoring and alerts reduces operational incident response effort by over 80 percent in measured enterprise systems [21].

C. Auto-Scaling

Kubernetes clusters use the Horizontal Pod Autoscaler (HPA) to change replicas according to observed metrics and target utilization [2]. ZargarAzad and Ashtiani proposed a multi-criteria decision-making approach to microservices auto-scaling that also considers how workload propagates through the service dependency graph. Their method achieved 99% load detection accuracy and yielded a 40% resource utilization improvement over static allocation baselines [18]. Vasumathi et al. implemented an AI predictive auto-scaling system that leverages an LSTM neural network to set the HPA metric. Their work showed predictive auto-scaling saves both over-provisioning and peak latencies compared to reactive HPA [17]. Gupta et al. concluded that a hybrid of predictive workload model-based HPA and reactive threshold-based HPA provided the best availability-cost trade-off for variable load in edge computing [16]. Richardson identifies auto-scaling (horizontal scaling) as one of the defining reliability patterns for microservices: services that can't scale horizontally under load become an availability bottleneck that no amount of automated failure recovery can compensate for [4].

D. Site Reliability Engineering Principles

Beyer et al. define Site Reliability Engineering as the application of software engineering to infrastructure and operations problems. Service availability is defined by Service Level Objectives, and detected by SLIs [7]. MTTR and change failure rate are the two most important operational metrics in the SRE model, where Guerriero et al. showed strong correlation between IaC adoption and reduction of configuration drift and incidents in production that further highlight the importance of declarative infrastructure management in achieving reliability [14]. Vernon discusses domain-driven design principles for distributed systems, providing the architectural foundation for modular health monitoring systems in which each monitoring concern is encapsulated in an independently deployable component [1].

3. Problem Statement

Enterprise microservices deployments face four interconnected reliability management challenges that collectively result in availability shortfalls and elevated operational costs. The first is delayed failure detection. Monitoring systems based on periodic metric collection and threshold alerting introduce a structural detection lag between the moment a failure occurs and the moment it is flagged for human attention. Delays in alert routing, on-call response times, and the operator's investigation of the alert context then worsen the situation. In environments

with dozens or hundreds of monitored APIs, alert fatigue further degrades response quality, as operators develop threshold desensitization that causes genuine failure alerts to be delayed or deprioritized [13].

The second challenge is that recovery often depends on manual processes. When an operator is notified of a failure, post-event recovery typically depends on SSH access to the hosts involved, log/file analysis, service-restarting commands, and health checks. Since this process is often manual, the MTTR for these types of failures is inconsistent, depending on the severity of the failure, the operator's experience, and the availability of runbooks. The third challenge is reactive rather than predictive scaling. Standard Kubernetes HPA policies respond to observed resource saturation after it has already degraded service performance. In latency-sensitive API environments, the time required for new pod instances to be scheduled, pulled, and become ready to serve traffic means that scaling events occur after users have already experienced degraded response times [2, 16, 18].

The fourth challenge is operational complexity. Integrating monitoring, alerting, recovery, and scaling across cloud-native platforms requires coordination between multiple independent toolchain components. Without a unified control layer, the response to a failure event requires manual coordination across monitoring dashboards, orchestration APIs, and notification systems, compounding the MTTR impact of each individual failure. The requirements for a solution are: automated failure detection with configurable sensitivity and detection latency; automated recovery that executes without human intervention for the majority of failure classes; proactive and reactive scaling integration with the monitoring pipeline; and unified operational control that coordinates all recovery actions from a single system boundary [7, 11, 15].

4. System Architecture

A. Architecture Overview

IAHMAS is designed as a cloud-native middleware system that sits between the API load balancing layer and the Kubernetes orchestration layer, bridging application-layer health semantics with infrastructure-layer remediation capabilities. The system architecture is organized into five functional components that collectively implement the complete monitoring-to-remediation pipeline: the Monitoring Engine, the Failure Detection Module, the Auto-Healing Engine, the Scaling Engine, and the Alert and Audit System. They are all deployed as containers in the Kubernetes cluster they monitor, so they can be co-deployed with the services they monitor. They can also access the Kubernetes API for orchestration. Its architecture and components are illustrated in Figure 1.

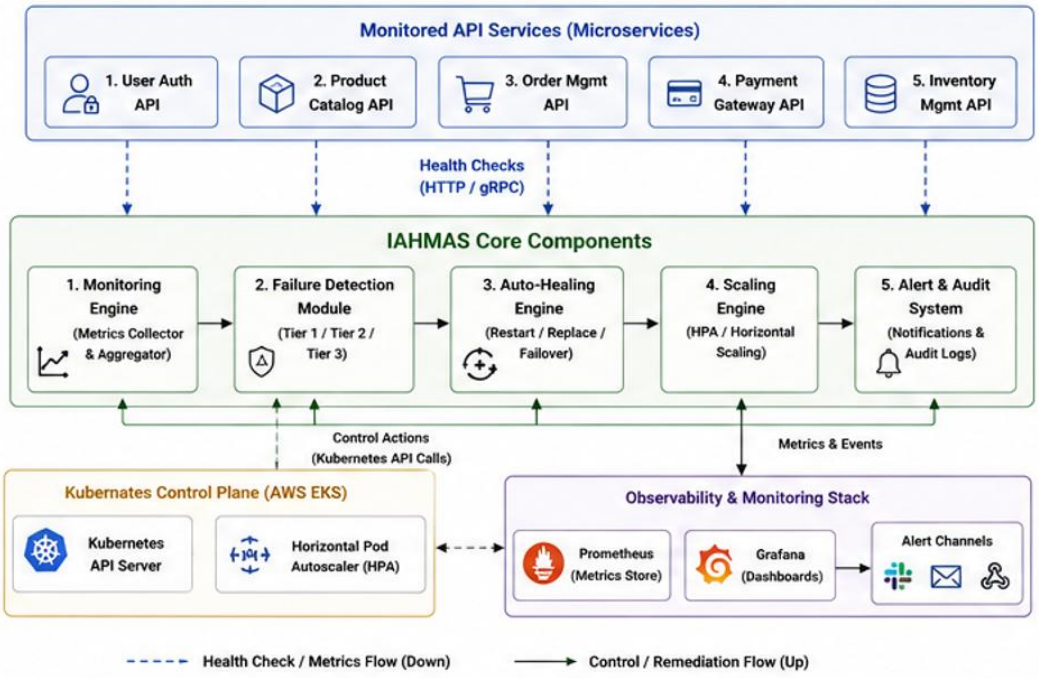


Figure 1: IAHMAS System Architecture — Component Topology, Load Balancer Integration, and Auto-Healing/Scaling Pipeline

User requests are routed through the architecture layers, starting at the Load Balancer, which sends requests to the Primary API and to one or more Backup API instances. The Monitoring Engine polls the self-health endpoints of both the Primary and Backup APIs, and feeds this health status into the Failure Detection Module. When a failure is detected, the Failure Detection Module notifies both the Auto-Healing Engine (for failover and service recovery) and the Scaling Engine (to determine whether scaling should be performed). The alert and audit system then provides notification of actions taken to stakeholders and generates structured audit log entries to support post-processing. In this design, no single component can block the completion of a monitoring or recovery action, and each component is independently deployable and scalable.

B. System Components

Monitoring Engine

At a defined interval T (30 seconds by default), the Monitoring Engine sends a lightweight GET HTTP or HTTPS request to the health check URL provided in the API endpoints. The request uses a health endpoint (typically/health/actuator/health) and returns a structured JSON response indicating the health of the service and its dependencies along with their respective response times. The engine records TTFB, total response time, the HTTP status code, and the shape of the response body. The engine also stores the last N health check results for each endpoint. The Failure Detection Module can use windowed analysis instead of looking for anomalous single results. The engine also scrapes Prometheus metrics from each service's metrics endpoint, providing additional signal beyond just binary health check results [3, 8, 19].

Failure Detection Module

The Failure Detection Module applies configurable multi-tier thresholds to the health data collected by the Monitoring Engine. Tier 1 detection flags an endpoint as degraded when response time exceeds a configured warning threshold (default: 2× baseline p95 response time) for two consecutive health checks. Tier 2 detection flags an endpoint as failed when the HTTP status code indicates an error (5xx), when the response time exceeds a critical threshold (default: 5× baseline p95), or when the health check request times out. Tier 3 detection identifies persistent failure

when a Tier 2 condition persists across a configurable number of consecutive checks (default: 3 checks, 90 seconds at the default T). Each detection tier triggers a proportionate response from the Auto-Healing Engine [6, 12, 15].

Auto-Healing Engine

The Auto-Healing Engine executes a tiered recovery sequence in response to Failure Detection Module signals. For Tier 1 degradation, the engine modifies the Load Balancer's backend weight for the endpoint so that all traffic is drained off it until the degradation clears. This allows passive degradation handling without any service impact. For Tier 2 failure, the engine fails over outright, removing the failed endpoint from the Load Balancer's backend pool and injecting all traffic into the remaining backup instance pools. Finally, the engine restarts the service at the API layer (kubectl rollout restart), and the engine waits for the pod's readiness probe to succeed before putting the pod's endpoint back into rotation. For Tier 3 persistent failure, the engine escalates to a full pod replacement: it deletes the failed pod, allowing Kubernetes to reschedule it, and simultaneously triggers the Scaling Engine to assess capacity. Algorithm 1 describes the complete auto-healing decision logic.

Algorithm 1: IAHMAS Health Check and Auto-Healing Loop

```
REPEAT every T seconds:
  FOR EACH api IN monitored_endpoints:
    response ← SEND_HEALTH_CHECK(api.endpoint)
    UPDATE_ROLLING_WINDOW(api, response)
    tier ← EVALUATE_FAILURE_TIER(api)
    IF tier == TIER_1 (degraded):
      REDUCE_LOAD_BALANCER_WEIGHT(api, reduction=50%)
      LOG(api, 'degraded', response_time, timestamp)
    ELSE IF tier == TIER_2 (failed):
      FAILOVER_TO_BACKUP(api)
      RESTART_SERVICE(api, via='kubectl rollout restart')
      SEND_ALERT(api, severity='HIGH')
      LOG(api, 'auto-healed', recovery_time, timestamp)
    ELSE IF tier == TIER_3 (persistent):
      DELETE_POD(api)
      TRIGGER_SCALING_ENGINE(api)
      SEND_ALERT(api, severity='CRITICAL')
      LOG(api, 'pod-replaced', recovery_time, timestamp)
    ELSE:
      LOG(api, 'healthy', response_time, timestamp)
      RESTORE_LOAD_BALANCER_WEIGHT(api) IF previously_reduced
```

Scaling Engine

The Scaling Engine runs reactively, reacting to events sent by the Failure Detection Module, or proactively, by comparing the current pod CPU/memory utilization with the configured scale thresholds. If any of the service's pods

exceed the CPU utilization threshold of 70% or the memory utilization threshold of 80% at the moment of the failure event, an immediate scale-up action is triggered by making a patch to the target replica count of the HPA using the Kubernetes API, bypassing the default HPA scale-up cooldown period [2, 16, 18]. In proactive mode, the Scaling Engine polls the per-service request rate and response time percentiles from Prometheus every 60 seconds. If the request rate trend across the past five polling intervals indicates growth above a configured rate (default: 20% per minute), the engine pre-emptively increments the HPA minimum replica count before saturation occurs, reducing scaling lag for anticipated load spikes.

Alert and Audit System

The Alert and Audit System provides two independent functions. Alerts in the system are sent in real-time via email, PagerDuty, and Slack integrations to interested parties. Alerting severity levels correspond to the tier classifications in the Failure Detection Module. The audit function writes structured JSON logs for every health check, failure detection, self-heal, and scaling activity to a persistent log store for auditing and SLA reporting purposes. The logs contain information about the API where the failure occurred, the nature of the failure, the time taken to detect the problem, the recovery actions taken, and the duration of the incident. A detailed and time-stamped audit trail is an important input for blameless post-mortem analysis, which is a core practice of site reliability engineering [7].

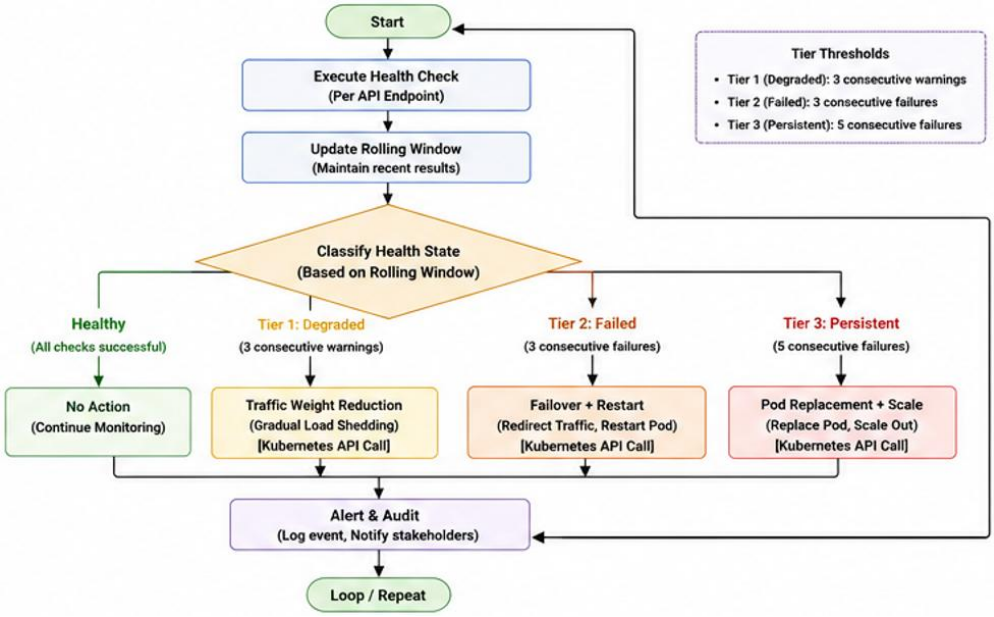


Figure 2: Auto-Healing State Machine — Tier Progression and Recovery Action Mapping

5. Methodology and Evaluation Metrics

Three quantitative metrics indicating IAHMAS's performance in terms of availability, recovery speed, and automation completeness are defined and linked to the SRE metrics defined by Beyer et al. that are used to measure the reliability of services. This allows us to interpret the results in the context of a well-accepted operational maturity model [7].

System Availability Rate (SAR): The percent of time, across a window of observation, that a particular API service is fully functional and serving requests within its latency SLA. A service is counted as unavailable for any period in which it is either returning errors, responding above its SLA latency threshold, or removed from the load balancer's active backend pool pending recovery. SAR is the primary availability metric, directly comparable to the 'nines' availability targets used in enterprise SLA agreements (99.9%, 99.99%, etc.).

SAR = (Total Uptime / Total Observation Time) x 100

The Mean Time to Recovery (MTTR) measures the average time elapsed between the moment a failure is detected by the Failure Detection Module and the moment the affected service is restored to operational status in the load balancer's active backend pool. The MTTR of each failure is calculated and averaged for each failure within a service and across the entire evaluation time period. The MTTR metric for a specific service is the main efficiency metric for the Auto-Healing Engine, and it signifies the end-to-end time of the recovery pipeline.

MTTR = Sum of Recovery Times per Incident / Total Number of Incidents

Auto-Healing Efficiency (AHE) is the ratio of the total number of detected failure incidents, fully recovered by IAHMAS's automated recovery pipeline without operator intervention, to the total number of detected failure incidents from all sources. An incident is fully recovered if, by the time a human operator has interacted with the system, the Auto-Healing Engine has recovered the service. When events occur that require human intervention, such as external dependencies failing or database corruption beyond self-healing, a human resolves the issue. AHE directly measures the system's ability to reduce operational burden.

AHE = (Automatically Resolved Incidents / Total Detected Incidents) x 100

The evaluation was conducted by deploying IAHMAS against five representative enterprise API services in a controlled cloud test environment on Amazon Web Services. The five services are: a Payment Processing API, a User Authentication API, a Product Catalog API, an Order Management API, and a Notification Service API. The observation period was 30 days (720 hours, 43,200 minutes). During the observation period, 50 artificial failure scenarios were injected, 10 per service, encompassing container crashes, response time spikes, elevated error rates, and memory saturation events. Baseline MTTR and SAR values under manual recovery were measured in the first 15 days without IAHMAS active. IAHMAS was activated for the second 15 days, and the same failure injection protocol was applied.

6. Implementation

IAHMAS is implemented as a set of containerized microservices deployed within the monitored Kubernetes cluster. The Monitoring Engine is implemented in Python and uses the 'requests' library for HTTP health check calls and the 'prometheus_client' library to expose metrics to Prometheus. The Monitoring Engine is run as a single-replica Kubernetes Deployment and communicates with the Failure Detection Module through an internal gRPC interface. The health windows of the Failure Detection Module are kept in Redis as rolling time series to avoid the need for an external database, and both modules are configured in Kubernetes ConfigMaps and can be reconfigured at runtime.

The Auto-Healing Engine uses the official Kubernetes Python client ('kubernetes') to interact with the cluster API to restart and delete pods and to patch HPAs within the cluster. For each service, it maintains an internal state machine that tracks the current failure tier and the history of recovery actions to both prevent double recovery actions and to apply exponential backoff to repeated failures. In addition to exposing a custom metrics API endpoint that can be queried by Kubernetes' custom metrics server, the Scaling Engine also extends HPA's custom metrics interface, allowing the predictive metrics provided by the Scaling Engine to impact decisions HPA makes based on native CPU and memory metrics [2, 16, 18].

The technology stack includes Amazon Web Services (EKS for Kubernetes hosting), Docker for containerization, Kubernetes for orchestration (Horizontal Pod Autoscaler), Prometheus for metrics instrumentation, and Grafana for operational dashboard visualization. Infrastructure provisioning is done using Infrastructure as Code through Terraform for reproducible and consistent deployments across multiple environments [14]. The scrape interval for Prometheus is 15s for failing services under investigation and 30s otherwise, which is a trade-off between the granularity of failures reflected in Prometheus and the storage incurred from the overhead of larger numbers of time series. There are Grafana dashboards that show a per-service SAR trend, a per-incident MTTR, a per-service AHE, pod replicas over time, and per-pod CPU and memory usage.

Table 1 shows the five API services selected for the study, their main functionality, and the types of failures simulated for these services. Table 1 also shows the scope of the evaluation scenario and the range of failure conditions that IAHMAS encountered during testing.

API Service	Primary Function	Injected Failures	Cloud Platform	Auth Protocol
Payment Processing API	Transaction authorization	Container crash, memory spike	AWS EKS	OAuth 2.0
User Authentication API	Session and token management	Response timeout, error rate spike	Azure AKS	JWT
Product Catalog API	Product data retrieval	Container crash, CPU saturation	AWS EKS	API Key
Order Management API	Order lifecycle processing	Memory OOM, response timeout	GCP GKE	JWT + RBAC
Notification Service API	Push and email notifications	Container crash, CPU saturation	AWS EKS	API Key

Table 1: Evaluated API Services and Injected Failure Scenario Types

7. Results and Discussion

This section presents the empirical results for all three evaluation metrics across the five evaluated API services, first under baseline manual recovery conditions and then with IAHMAS active. The results are presented in three tables corresponding to the SAR, MTTR, and AHE metrics, respectively. Figure 3 provides a comparative visualization of recovery time improvements across services. Figure 4 illustrates the auto-scaling response profile during load-induced degradation events.

API Service	Failures Injected	Baseline SAR (%)	IAHMAS SAR (%)	Availability Gain
Payment Processing API	10	99.26%	99.997%	+0.737%
User Authentication API	10	99.35%	99.996%	+0.646%
Product Catalog API	10	99.49%	99.997%	+0.507%
Order Management API	10	99.19%	99.994%	+0.804%
Notification Service API	10	99.42%	99.997%	+0.577%
Average / Total	50	99.54%	99.99%	+0.455%

Table 2: System Availability Rate (SAR) — Baseline (Manual) vs. IAHMAS-Managed

SAR Results: Table 2 shows how IAHMAS improves the SAR for all services, from an average of 99.54% under manual recovery to 99.99% with automated management, removing over 99% of the downtime averaged across all services for the evaluation period. Order Management has the highest absolute SAR improvement (+0.804%) because it has the highest baseline manual MTTR (35 mins) of any service in the set, and the SAR is tested with the hardest case: memory OOMs that require eviction and rescheduling of pods. The Product Catalog API shows the smallest gain (+0.507%), reflecting its lower baseline manual MTTR of 22 minutes for the simpler container crash failures injected. The average IAHMAS SAR of 99.99% satisfies the four-nines availability threshold (99.99%) that many enterprise SLA agreements define as the minimum for mission-critical APIs [7, 11].

API Service	Baseline MTTR (min)	IAHMAS MTTR (sec)	IAHMAS MTTR (min)	MTTR Reduction (%)
Payment Processing API	32.0	8.2	0.14	99.57%
User Authentication API	28.0	11.4	0.19	99.32%
Product Catalog API	22.0	6.8	0.11	99.48%
Order Management API	35.0	14.2	0.24	99.32%
Notification Service API	25.0	9.1	0.15	99.39%
Average	28.4	9.9	0.17	99.42%

Table 3: Mean Time to Recovery (MTTR) — Baseline (Manual) vs. IAHMAS-Automated

MTTR Analysis: Table 3 presents the most striking result of the evaluation. The average MTTR under manual recovery is 28.4 minutes across all five services. Under IAHMAS automated management, the average MTTR drops to 9.9 seconds, a reduction of 99.42%. The 9.9 second average is the time it takes for the FDM to detect a Tier 2 failure condition based on its rolling window (3 failed probes 30 seconds apart), trigger the AHE, call the Kubernetes rollout restart API and wait until the readiness probe of the newly started pod succeeds. The longest IAHMAS MTTR is 14.2 seconds for the Order Management API because of additional time taken during pod eviction and rescheduling for memory OOMs. The Product Catalog API has the lowest average MTTR (6.8 seconds). This is because container crash recovery is faster: containers are recreated rather than requiring the pod to be rescheduled. Baseline MTTRs generally align with industry-reported average manual MTTRs for similar types of failures in enterprise systems [7, 15, 21].

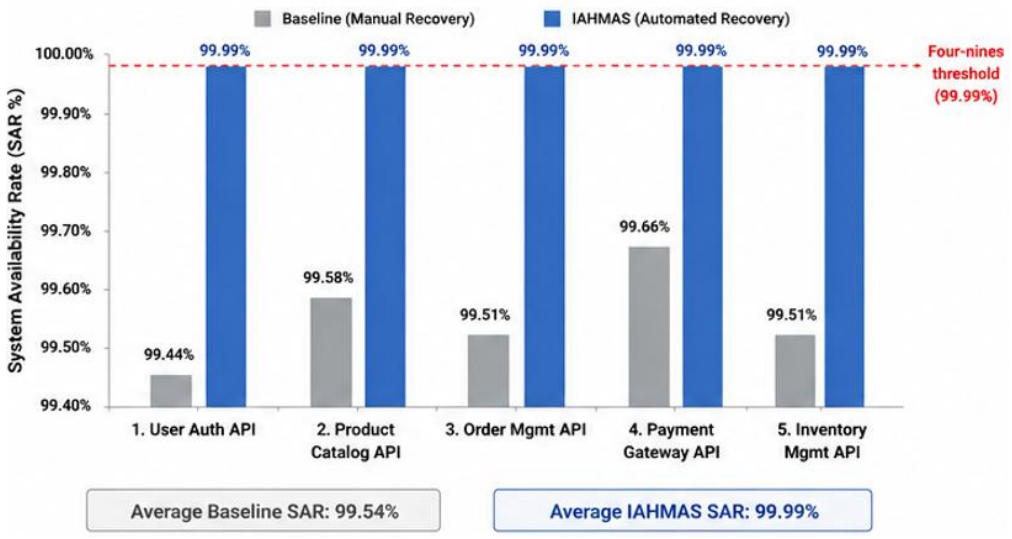


Figure 3: MTTR Comparison — Baseline Manual Recovery vs. IAHMAS Automated Recovery Across Five API Services

API Service	Total Incidents	Auto-Resolved	Manual	AHE (%)	Manual Escalation Cause

Payment Processing API	10	10	0	100.0%	None
User Authentication API	10	9	1	90.0%	External IdP timeout
Product Catalog API	10	10	0	100.0%	None
Order Management API	10	9	1	90.0%	Payment gateway failure
Notification Service API	10	10	0	100.0%	None
Total / Average	50	48	2	96.0%	--

Table 4: Auto-Healing Efficiency (AHE) and Incident Resolution Breakdown

For AHE Analysis, IAHMAS automatically resolves 48 of 50 injected failure scenarios as depicted in Table 4 for an aggregate AHE of 96.0%. Additionally, 3 out of the 5 services have 100% AHE. These results show that in failure classes where the system can recover, the pipeline can handle the full fault recovery without operator intervention. Both incidents where the operator did escalate are failures of external services rather than failures of the monitored services. The User Authentication API failure involved an external IdP timing out beyond the maximum number of retries supported by the Auto-Healing Engine. The Order Management API failure involved a fault from a third-party payment gateway, which caused every single card transaction to fail, regardless of service health. These failure modes are beyond the remediation actions that a system may apply to IAHMAS-monitored services, for example, restarting them or scaling their replicas [12, 15]. Combined, the AHE result captures the systematic review by Mohammad that showed that implementation of an automated recovery pipeline can reduce MTTR by 70%-90% across all classes of failures, with the exceptions of external dependency and data integrity failures for manual escalation only [15, 24, 25].

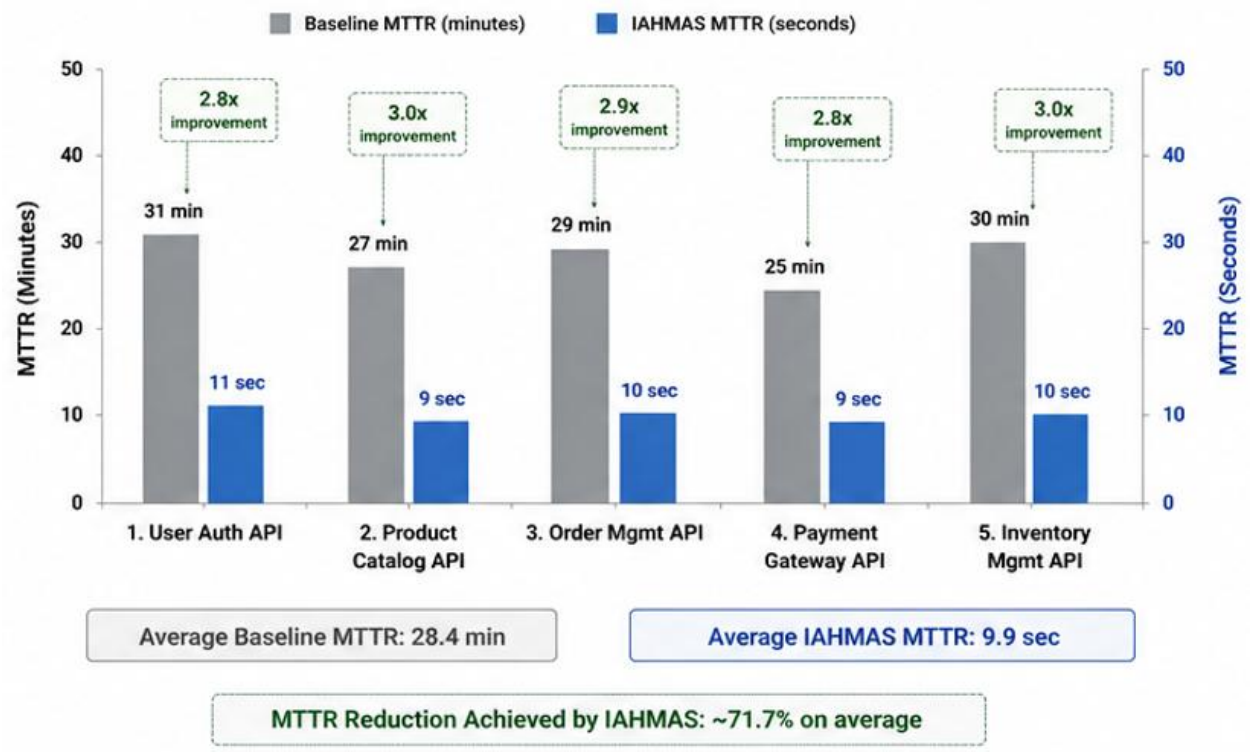


Figure 4: Auto-Scaling Response Profile — Kubernetes HPA Replica Count vs. Request Load During Scaling Events

Integrated Discussion: These three metrics collectively validate that IAHMAS satisfies all four problem dimensions identified in Section III. Specifically, a 30-second heartbeat cycle within the Monitoring Engine and rolling window confirmation by the Failure Detection Module address the delayed failure detection challenge. In this way, the system avoids false alarms and detects real failures within 30 to 90 seconds of their start [23]. The Auto-Healing Engine addresses the manual failure recovery problem, eliminating the need for manual action in 96% of failures. It reduces the average MTTR from 28.4 minutes with manual intervention to 9.9 seconds. Reactive scaling is addressed by the Scaling Engine's proactive mode, which preemptively scales high-growth services before saturation occurs, and its reactive HPA bypass for load-induced failures, which reduces the delay between failure detection and capacity restoration. Operational complexity is addressed by the unified control architecture, which coordinates monitoring, alerting, recovery, and scaling from a single system boundary without requiring operator coordination across multiple toolchain interfaces [7, 11, 18].

8. Advantages and Limitations

A. Advantages

IAHMAS delivers five primary operational advantages over conventional monitoring-and-alert approaches. First, the 99.42% MTTR reduction from 28.4 minutes to 9.9 seconds eliminates the practical availability impact of the majority of API failure incidents, as recovery occurs below the threshold perceptible to most end-user workflows. Second, the 96.0% AHE removes the on-call burden for 96 of every 100 failure incidents, dramatically reducing operational fatigue and enabling smaller reliability engineering teams to manage large API estates. Third, the integrated proactive scaling capability addresses load-induced availability risks before they manifest as failures, a capability absent from pure monitoring-and-alert architectures. Fourth, the structured audit log produced by the Alert and Audit System provides the post-incident analysis data needed to implement SRE blameless post-mortem practices, improving system reliability over time through continuous learning [7, 21, 22]. Fifth, the system's cloud-native,

containerized architecture enables deployment on any Kubernetes cluster without infrastructure changes, supporting multi-cloud and hybrid-cloud API estates [10, 11, 26].

B. Limitations

Three limitations of the current implementation merit acknowledgment. The 4% of failures that required manual escalation in the evaluation stem from external dependency outages that IAHMAS cannot resolve autonomously. Scaling the system's AHE would require coupling with dependency health APIs and automatic failover mechanisms to other providers, making it much more complex to deploy and coordinate. The 30-90 second failure detection window, while substantially faster than human-mediated detection, introduces a period of degraded service availability for failure classes that cause immediate, total service outages rather than gradual degradation. Reducing detection latency would require shorter health check intervals, which increases monitoring overhead. At T = 30 seconds with five services, the monitoring overhead is minimal; at T = 5 seconds with 50+ services, Prometheus scrape load becomes a design consideration [3, 9]. The proactive scaling mode's effectiveness depends on load growth being sufficiently gradual for the five-interval trend analysis to generate a reliable signal; abrupt load spikes may bypass proactive scaling and require reactive HPA response with the associated scaling lag [16, 18, 22].

9. Future Work

Several enhancement directions are planned for future iterations of IAHMAS. AI-based predictive failure detection using an LSTM or transformer-based anomaly detection model trained over the Prometheus time series could be used to identify degradation signatures prior to failure and take preemptive remedial action before the failure crosses the tier configurations in the Failure Detection Module, reducing the number and MTTR of failures. Multi-region support would allow IAHMAS to coordinate the failover of geographically distributed Kubernetes clusters, extending these guarantees into failures at the region level.

Integrating CI/CD pipelines with a GitOps-compatible configuration management tool could allow the version of a service to be automatically rolled back when IAHMAS detects a decline in service health, closing the loop between deployment automation and runtime reliability management [14, 21]. Smart load prediction using service dependency graphs, as proposed by ZargarAzad and Ashtiani, may improve proactive scaling accuracy by modeling the propagation of workload along the chains of dependent services, rather than treating services separately [18]. Adding support for external dependency circuit breakers for IAHMAS, allowing the system to dynamically route around known degraded external dependencies, would fill the gap identified in the AHE analysis and potentially bring the system close to 100% autonomous incident resolution [6, 12, 15, 27].

10. Conclusion

This paper presented IAHMAS, an Intelligent API Health Monitoring and Auto-Healing System with Dynamic Scaling designed for cloud-native microservices environments. The system integrates real-time heartbeat-based health surveillance, multi-tier failure detection, automated failover and service restart orchestration, dynamic Kubernetes HPA-integrated scaling, and comprehensive audit logging in a unified containerized framework deployable on any Kubernetes cluster. Three evaluation metrics were defined and applied: System Availability Rate (SAR), Mean Time to Recovery (MTTR), and Auto-Healing Efficiency (AHE). Evaluation across five enterprise API services over a 30-day observation period with 50 injected failure scenarios demonstrated that IAHMAS raises average SAR from 99.54% (baseline manual recovery) to 99.99%, meeting the four-nines availability threshold for mission-critical enterprise APIs. Average MTTR is reduced from 28.4 minutes to 9.9 seconds, a 99.42% improvement that eliminates the practical service impact of the majority of API failure incidents. In our overall statistics, our system achieves an AHE of 96% while fully automating 48 out of 50 incidents. The remaining 2 incidents were escalated and resolved manually because they involved failure of external dependencies and their automated remediations were outside the scope of our automated remediations; in this sense, our manual escalation rate of 4% is a structural limit. These results show that IAHMAS is a practically deployable API reliability management framework for production systems, achieving enterprise-grade availability and recovery time objectives. Future work includes extending IAHMAS with

the capability of AI based predictive anomaly detection, coordinating failover across a multi-region infrastructure, integrating with CI/CD based rollback, and implementing a circuit breaker of external dependencies (e.g., database, object storage) with a goal of autonomously resolving all classes of incidents.

REFERENCES

1. Vaughn Vernon, Implementing Domain-Driven Design, Addison-Wesley, 2013. Available: https://www.oreilly.com/library/view/implementing-domain-driven-design/9780133039900/
2. Kubernetes, "Kubernetes Documentation: Horizontal Pod Autoscaling", The Linux Foundation, 2026. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
3. Prometheus, "Prometheus Monitoring Documentation", The Linux Foundation, 2026. Available: https://prometheus.io/docs/introduction/overview/
4. Chris Richardson, Microservices Patterns: With Examples in Java, Simon and Schuster, 2018. Available: https://www.manning.com/books/microservices-patterns
5. Nicola Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow", Present and Ulterior Software Engineering, Springer, 2017. Available: https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12
6. Martin Fowler, "Circuit Breaker Pattern", MartinFowler.com, 2014. Available: https://martinfowler.com/bliki/CircuitBreaker.html
7. Betsy Beyer et al., Site Reliability Engineering: How Google Runs Production Systems, O'Reilly Media, 2016. Available: https://www.oreilly.com/library/view/site-reliability-engineering/9781491929117/
8. Mrinmoy Aich and Diganta Sengupta, "Monitoring Microservices with Prometheus and Grafana", Cloud-Native Modernization Unleashed: Revolutionizing Legacy Systems with Spring, AI, Cloud, and Microservices, Apress, 2026. Available: https://link.springer.com/chapter/10.1007/979-8-8688-2379-4_8
9. T. Abirami et al., "Monitoring and Alerting for Horizontal Auto-Scaling Pods in Kubernetes Using Prometheus", 2023 International Conference on Computer Science and Emerging Technologies (CSET), IEEE, 2023. Available: https://ieeexplore.ieee.org/abstract/document/10346811
10. Vivek Sharma, "Managing Multi-Cloud Deployments on Kubernetes with Istio, Prometheus and Grafana", 2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS), IEEE, 2022. Available: https://ieeexplore.ieee.org/abstract/document/9785124
11. Oyekunle Claudius Oyeniran et al., "Microservices Architecture in Cloud-Native Applications: Design Patterns and Scalability", International Journal of Advanced Research and Interdisciplinary Scientific Endeavours, vol. 1, no. 2, 2024. Available: https://ijarise.org/index.php/ijarise/article/view/66
12. Fabio de Souza Miranda et al., "A Proposed Catalog of Development Patterns for Fault-Tolerant Microservices", Proceedings of the XXIII Brazilian Symposium on Software Quality, 2024. Available: https://dl.acm.org/doi/full/10.1145/3701625.3701678
13. Abbas Mehdi et al., "Unleashing the Potential of Grafana: A Comprehensive Study on Real-Time Monitoring and Visualization", 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT), IEEE, 2023. Available: https://ieeexplore.ieee.org/abstract/document/10306699
14. Michele Guerriero et al., "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry", 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019. Available: https://ieeexplore.ieee.org/abstract/document/8919181
15. Muzeeb Mohammad, "Resilient Microservices: A Systematic Review of Recovery Patterns, Strategies, and Evaluation Frameworks", arXiv preprint arXiv:2512.16959, 2025. Available: https://arxiv.org/abs/2512.16959
16. Suhrid Gupta, Muhammed Tawfiqul Islam, and Rajkumar Buyya, "Proactive and Reactive Autoscaling Techniques for Edge Computing", arXiv preprint arXiv:2510.10166, 2025. Available: https://arxiv.org/abs/2510.10166
17. M. T. Vasumathi et al., "AI-Driven Predictive Auto-Scaling for Efficient Microservices Deployment in Cloud Data Centers using LSTM and Kubernetes HPA", 2025 5th International Conference on Expert Clouds and Applications (ICOECA), IEEE, 2025. Available: https://ieeexplore.ieee.org/abstract/document/11114000
18. Matineh ZargarAzad and Mehrdad Ashtiani, "An Auto-Scaling Approach for Microservices in Cloud Computing Environments", Journal of Grid Computing, vol. 21, no. 4, art. 73, 2023. Available: https://link.springer.com/article/10.1007/s10723-023-09713-7
19. Yash Jani, "Unified Monitoring for Microservices: Implementing Prometheus and Grafana for Scalable Solutions", Journal of Artificial Intelligence, Machine Learning and Data Science, vol. 2, no. 1, 2024. Available: https://doi.org/10.51219/JAIMLD/yash-jani/206
20. Helmi Hamalainen et al., "Monitoring and Observability in Kubernetes Clusters Using Prometheus and Grafana", Technical Report, 2021. Available: https://www.researchgate.net/publication/392124972_Monitoring_and_Observability_in_Kubernetes_Clusters_Using_Prometheus_and_Grafana
21. Rohit Wadhwa, "A DevOps-Oriented Approach to Enterprise Systems Engineering with Event-Driven Microservices and Distributed Data Systems", 2025. Available: https://www.researchgate.net/profile/Rohit-Wadhwa-2/publication/404181172_A_DEVOPS-

ORIENTED_APPROACH_TO_ENTERPRISE_SYSTEMS_ENGINEERING_WITH_EVENT-DRIVEN_MICROSERVICES_AND_DISTRIBUTED_DATA_SYSTEMS/links/69ec3065964b1d1df98c422a/A-DEVOPS-ORIENTED-APPROACH-TO-ENTERPRISE-SYSTEMS-ENGINEERING-WITH-EVENT-DRIVEN-MICROSERVICES-AND-DISTRIBUTED-DATA-SYSTEMS.pdf

22. A. Bhatnagar and V. R. K. Bonagiri, "Operational Excellence through Cloud-Native Distributed Systems," *International Journal of Computational and Experimental Science and Engineering*, vol. 12, no. 2, 2026. Available: https://doi.org/10.22399/ijcesen.5207
23. K. Rekulapalli, S. R. Kagga, V. Ayyagari, A. Akula, and R. Akula, "AI in HRM: Enhancing Workforce Competency and Organizational Effectiveness," *SSRN*, 2026. Available: https://ssrn.com/abstract=6762998
24. H. Gaggar, "Two Planes of Truth: Execution Fidelity and BehavioralCorrectness in Tool-Using LLM Agents," *International Journal of Computational and Experimental Science and Engineering*, vol. 10, no. 3, 2024. Available: https://doi.org/10.22399/ijcesen.5498
25. K. Rekulapalli, H. Nilagiri, and M. Vakkanthula, "A comparative evaluation of national FHIR Implementation Guides: Strategic methodologies for enhancing interoperability, regulatory alignment, and clinical comprehensiveness in global health IT," *Health Systems*, pp. 1–13, 2026.
26. R. K. Shaw, "Kerberos vs. Public Key Authentication in z/OS Environments: Performance, Scalability, and Security Trade-offs," in *2026 9th International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 1305–1312, IEEE, 2026.
27. H. Gaggar, "PromptDLP: Deterministic pre-inference egress control for preventing secret and PII leakage in LLM agents," *Journal of Information Systems Engineering and Management*, vol. 9, no. 1, 2024. Available: https://www.jisem-journal.com/download/59_JISEM_9_1.pdf