AdaptiFlow: An Extensible Framework for Event-Driven Autonomy in Cloud Microservices

Brice Arléon Zemtsop Ndadji
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
brcie-arleon.zemtsop-ndadji@inria.fr

Simon Bliudze
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
simon.bliudze@inria.fr

Clément Quinton
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
clement.quinton@inria.fr

Modern cloud architectures demand self-adaptive capabilities to manage dynamic operational conditions, yet existing solutions often impose centralized control models ill-suited to microservices' decentralized nature. This paper presents **AdaptiFlow**, a framework that reimagines autonomic computing through abstraction layers focused on the **Monitor** and **Execute** phases of the MAPE-K loop [7]. By decoupling metrics collection and action execution from adaptation logic, AdaptiFlow enables microservices to evolve into autonomous elements through standardized interfaces, preserving their architectural independence while enabling system-wide adaptability. The framework introduces: (1) **Metrics Collectors** for unified infrastructure/business metric gathering, (2) **Adaptation Actions** as declarative actuators for runtime adjustments, and (3) a lightweight **Event-Driven** and rule-based mechanism for adaptation logic specification. Validation through the enhanced **Adaptable TeaStore** benchmark [2] demonstrates practical implementation of three adaptation scenarios targeting three levels of autonomy—self-healing (database recovery), self-protection (DDoS mitigation), and self-optimization (traffic management)—with minimal code modification per service. Key innovations include a workflow for gradual service instrumentation and evidence that decentralized adaptation can emerge from localized decisions without global coordination. The work bridges autonomic computing theory with cloud-native practice, providing both a conceptual framework and concrete tools for building resilient distributed systems. Future work includes integration with formal coordination models like JavaBIP [3] to avoid conflicting adaptation scenarios and improved AI agents techniques using AWARE [16] for proactive adaptations.

Keywords: self-adaptive systems, cloud microservices, MAPE-K loop, decentralized adaptation, autonomic computing, adaptive workflows

1 Introduction

Modern cloud architectures face growing complexity due to their distributed nature, necessitating systems that autonomously adapt to dynamic conditions. The MAPE-K loop [7] (Monitor-Analyze-Plan-Execute-Knowledge) has long served as the foundation for self-adaptive systems, traditionally implemented as a centralized, reactive, and sequential loop for executing adaptations [16]. However, microservices' decentralized nature demands a paradigm shift toward separating functional and adaptation concerns. Drawing inspiration from hardware abstraction layers (HALs) in operating systems, this paper

Submitted to:
WACA 2025

introduces an abstraction layer focused on the **Monitor** and **Execute** phases of MAPE-K, enabling microservices to expose monitoring data and accept control commands without invasive code changes. As envisioned in the original Autonomic Computing Vision (ACV) [7], this approach ensures "the managed element can be adapted to enable the autonomic manager to monitor and control it," bridging the gap between non-adaptive systems and self-managing elements.

We present **AdaptiFlow**, a framework providing standardized interfaces to instrument microservices for self-adaptation. AdaptiFlow's architecture is designed to address the unique challenges of self-adaptive systems in cloud-native environments, where decentralization, scalability, and context-awareness are critical. Self-adaptive systems are capable of dynamically altering their structure and behavior during runtime by continuously evaluating their environment, internal state, and operational goals [11]. For instance, consider a microservice that responds to a sudden workload increase by disabling non-essential functionalities. An online retailer, for example, might temporarily deactivate its computationally intensive recommendation engine during peak traffic periods. Such adaptations enable the system to preserve critical performance metrics (e.g., latency, cpu / memory usage) despite fluctuating demands. To build a self-adaptive system, developers must encode logic that defines the data to observe (e.g. performance metrics), the events (e.g. traffic increase, service unavailable, DDoS Attack), the conditions triggering adaptation and the specific mechanisms for executing those changes. AdaptiFlow's core abstraction layer comprises:

- **Metrics Collectors**: Unified APIs to gather infrastructure (CPU, latency) and business-level metrics (cache hits, transaction rates).
- **Adaptation Actions**: Declarative interfaces to define infrastructure-level (e.g., scaling) and business-level (e.g., feature toggling) actuators.

By decoupling monitoring and execution from adaptation logic, AdaptiFlow enables diverse strategies for analysis and planning. Developers can implement:

- **Internal Logic**: Rule-based adaptations (e.g., threshold-driven events) embedded directly into services.
- **Exogenous Logic**: External agents or planners (e.g., AI agents) leveraging AdaptiFlow's interfaces to collect metrics and execute actions.

To validate this approach, we augment AdaptiFlow with event-driven concepts that enable rule-based adaptation logic. The framework provides *Event Observation* mechanisms to detect threshold-based (e.g., CPU > 80%) or custom events (e.g., service degradation patterns) through periodic polling or on-demand triggers. These events act as semantic bridges between raw metrics and actionable adaptations, allowing developers to declaratively specify *when* and *how* the system should respond to changing conditions.

We validate AdaptiFlow by building the *Adaptable TeaStore* with the main support of the adaptation scenarios described in the *Adaptable TeaStore Specification* [2], a description of the TeaStore microservices benchmark [18] extended to support autonomous behaviour. Our experiments demonstrate AdaptiFlow's practicality through three implemented adaptation scenarios targeting three levels of autonomy: self-healing (service recovery), self-protection (DDoS mitigation), and self-optimization (traffic management) with minimal code changes.

Key Contributions:

- An abstraction layer for monitoring/execution, transforming non-adaptive services into autonomic elements.

- A workflow to instrument microservices with metrics collectors and actuators, aligning with ACV's vision.
- Empirical validation through the *Adaptable TeaStore* [2], an extended microservice benchmark supporting autonomic behaviors.

Paper Structure: Section 2 present the state of art, Section 3 outlines the framework design. Section 4 details experimental results with the TeaStore [18] case study. Section 5 concludes with future work, including integration with formal coordination models. A class diagram is provided in the appendix.

2 State of art

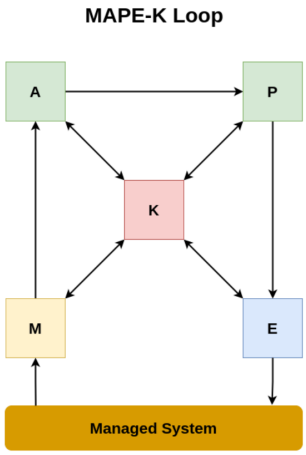

Figure 1: MAPE-K Loop with all functional elements: Monitor, Analyze, Plan, Execute, Knowledge.

The evolution of self-adaptive systems has been shaped by three fundamental paradigms that influence the design of AdaptiFlow: the principles of autonomic computing [7] , the MAPE-K control loop [7] and rule-based adaptation techniques [15]. These fundamental concepts are progressively addressing the challenges of managing complex distributed systems, from early monolithic architectures to modern cloud-native environments. Figure 1 provides a visual summary of the MAPE-K reference model that underpins much contemporary adaptation research.

2.1 Autonomic Computing and Self-Adaptive Systems

Modern computing systems are evolving toward autonomic paradigms, where components manage their behavior autonomously to reduce human intervention. Inspired by biological systems, the vision of autonomic computing [7] envisions distributed networks of *autonomic elements*, self-managing entities that deliver services while adhering to predefined policies. Each element consists of a *managed resource* (e.g., a database, CPU, or microservice) and an *autonomic manager* that monitors and controls it. Over time, the distinction between manager and managed resource may dissolve [7], yielding fully integrated elements capable of independent decision-making while collaborating through decentralized interactions. This shift mirrors natural systems like ant colonies, where collective intelligence emerges from localized interactions [7] rather than centralized control.

2.2 MAPE-K Framework Fundamentals

The MAPE-K (Monitor-Analyze-Plan-Execute-Knowledge) loop (Fig. 1), introduced over two decades ago by Kephart et al. [7], remains a cornerstone of self-adaptive systems. It structures adaptation into four phases:

- **M**onitor: Observes the system's internal state and environment (e.g., CPU usage, request latency).
- **A**nalyze: Evaluates observations to identify adaptation needs (e.g., detecting service failures).
- **P**lan: Generates strategies to achieve desired states (e.g., scaling instances, enabling circuit breakers).
- **E**xecute: Implements selected adaptation actions.

A shared **K**nowledge (K) repository stores contextual data (e.g., metrics, policies) to support decision-making across phases. Originally applied to monolithic systems, MAPE-K has since been adapted for cloud computing [9, 6, 11], IoT [12, 13], and cybersecurity [4, 5, 17].

2.3 Rule-Based Adaptation with Drools

Rule-based systems like Drools [15] can be use to operationalize the **Analyze** and **Plan** phases of MAPE-K by codifying adaptation logic into declarative condition-action pairs. A rule's structure "when *condition*, then *action*" enables developers to declaratively specify behaviors (e.g., "when CPU > 80%, then scale instances"). Unlike traditional code, rules are modular, auditable, and dynamically updatable, making them ideal for scenarios requiring rapid adjustments (e.g., fraud detection, load management). For example, Drools filters data through conditions and triggers actions when matches occur. This approach bridges technical and business requirements, allowing non-developers to contribute to adaptation policies.

While there exist frameworks, e.g., rule engines like Drool, that simplify adaptation logic, we are not aware of lightweight solutions providing a common interface for service monitoring and execution and focusing on the separation of concerns between the functional and adaptation layers in Cloud Computing systems.

3 Framework Design

AdaptiFlow's architecture reimagines autonomous management for cloud-native systems through a decomposition of the MAPE-K loop. Our design introduces a modular approach based on two key principles: (1) standardization of observation and actuation interfaces, and (2) flexible integration of various adaptation strategies. This separation allows microservices to retain their architectural independence while participating in system-wide adaptation models. Figure 2 provides a visual illustration of this decentralized architecture, which we detail in the following subsections.

3.1 Architectural Overview

AdaptiFlow provide a set of directives to specify the adaptation logic. The framework operationalizes the MAPE-K loop by focusing on two core phases **Monitor** and **Execute** tailored for decentralized cloud environments. We intentionally omit the **Analyze** and **Plan** phases since our goal is to allow the separation of concerns. However, in order to allow closing the loop in the absence of these two phases, we do provide the mechanism for triggering actions when a corresponding condition is satisfied using

an event-driven methodology. Since our solution is provided in the form of a Java library, it allows for specification of arbitrary custom adaptation strategies. However it intentionally does not provide any dedicated syntax or abstractions for that purpose. These are left for separate future work.

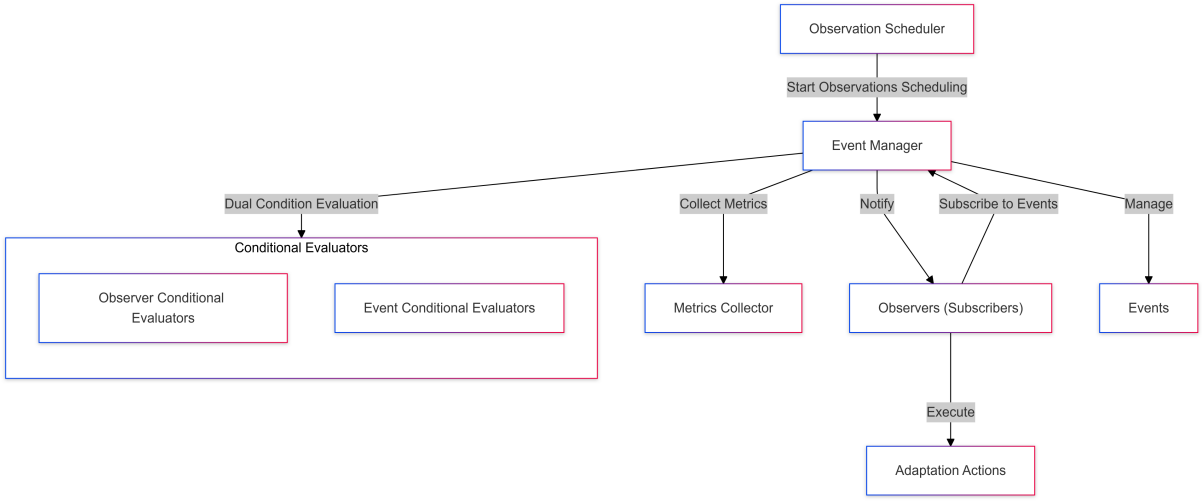

Figure 2: AdaptiFlow's decentralized architecture, emphasizing modular components for metrics collection, adaptation action execution and event handling.

In the **Monitor** phase, metrics collectors continuously gather infrastructure (e.g., CPU, memory) and business-level (e.g., request rates, services states) data, providing real-time insights into system state. For example, a `LatencyCollector` might track API response times, while a `ResourceUsageCollector` monitors cloud resource availability.

The **Execute** phase then invokes predefined or custom adaptation actions, such as scaling services via Kubernetes (`ScaleService`) or disabling non-critical features like recommendations (EnableFallbackAlgorithm). AdaptiFlow use a decentralize decision-making through event subscriptions: services subscribe to relevant events (e.g., an auth service subscribing to `DDoSAttackEvent`), eliminating reliance on a global planner.

The **Analyze** phase employs conditional evaluators to interpret the data collected, detecting events such as `HighWorkloadEvent` (request rate > 1,000/s OR CPU > 80%) or `ServiceFailureEvent` (health check timeout). These evaluators apply threshold-based rules (e.g., CPU > 80%) or custom logic (e.g., correlating disk I/O with checkout failures) to determine adaptation triggers.

The shared **Knowledge** (K) is derived dynamically from distributed metrics rather than a static repository, ensuring context-aware decisions. This approach offers three key advantages: (1) *Decentralized control*, enabling parallel adaptations; (2) *Transparency*, as rules and events are explicitly defined and auditable; and (3) *Flexibility*, supporting hybrid adaptations that combine infrastructure adjustments (scaling) with business logic changes (feature toggling). AdaptiFlow distributes adaptation logic across microservices, aligning with their autonomous nature. This design choice ensures minimal overhead while enabling granular, context-driven adaptations.

To realize this architecture, AdaptiFlow decomposes the adaptation process into six modular components (Fig. 2), each addressing a distinct aspect of self-adaptation:

- **Metrics Collectors** (Perception of the Environment): gather infrastructure and business-level metrics (e.g., CPU usage, API latency) to establish a real-time view of the context.

- **Adaptation Actions** (Executing Change): translate decisions into runtime adjustments, ranging from infrastructure operations (e.g., scaling) to business logic modifications (e.g., enabling fallback algorithms).
- **Event Management**: the events are specify using conditional evaluators, then services or service components subscribe to them and the event observation start. When the conditions specify in the event conditional evaluator are meet, the event is trigger and the event subscribers are notify.
  - **Conditional Evaluators** (Adaptive Decision-Making): apply threshold-based or custom logic (e.g., "IF cache hit ratio < 20% AND peak hour") to determine when events should trigger and also when subscribers should be notify to execute adaptations. Conditional Evaluators are part of the specification of both events and events subscribers.
  - **Event Specification** (Contextual Awareness): defines the structure of an event by the use of conditional evaluators on the collected metrics (e.g., `ResourceExhaustionEvent` = CPU > 90% && FreeDisk < 10%). Developers implicitly specify event types (threshold-based or custom) and their triggering criteria, enabling precise alignment with adaptation goals.
  - **Event Subscription**: enables services or service components to declaratively register interest in specific events (e.g., a billing service subscribing to `HighErrorRateEvent`).
  - **Event Observation** (Contextual Awareness): Implements strategies to detect specified events, such as periodic polling (e.g., check CPU every 30 seconds) or event-driven triggers (e.g., API failure webhook). This component ensures timely responses to gradual trends (e.g., memory leaks) and sudden anomalies (e.g., DDoS attacks).

The following sections detail how these components collectively enable granular, context-aware adaptations while adhering to microservices' decentralized ethos. By decoupling data collection, event detection, and action execution, AdaptiFlow allows developers to specify and gradually extend the adaptation logic without making major architectural changes.

3.2 Metrics Collectors: Perception of the Environment

The foundation of self-adaptation lies in accurate perception. AdaptiFlow's *Metrics Collectors* are designed to gather both infrastructure-level (e.g., CPU, memory) and business-level metrics (e.g., transaction success rates, cache hit ratios). This dual focus ensures adaptations account for technical constraints and domain-specific requirements. For example, a `LatencyCollector` might monitor API response times, while a `CartAbandonmentCollector` tracks user behavior. Collectors enable services to expose their context for adaptation purposes. Once the collectors have been defined, it's easy to support both pull mechanisms (e.g. Prometheus) and push mechanisms (e.g. REST webhooks), accommodating diverse monitoring ecosystems. By decoupling data collection from analysis, AdaptiFlow allows developers to incrementally instrument services without overhauling existing systems.

3.3 Adaptation Actions: Executing Change

Adaptation actions translate decisions into runtime changes. AdaptiFlow enable the specification of adaptation actions that impact the:

- **Infrastructure-Level**: Platform-specific or DevOps operations like `ScaleService` or `RestartPod`.
- **Business-Level**: Domain-specific adjustments such as `EnableFallbackRecommender` (switching to a lightweight algorithm) or `EnableCache`.

Actions execute either synchronously (e.g., immediate circuit breaking) or asynchronously (e.g., batched log cleanup). Using our provided mechanism for adaptation logic specification, developers bind actions to events subscribers (services or service components). Each subscriber define a list of adaptation actions that will be executed when the conditions define in both the event and the subscriber will be met. For example, a billing service might subscribe to `HighErrorRateEvent` to disable premium features temporarily, while a load balancer scales instances for the same event.

3.4 Conditional Evaluators: Adaptive Decision-Making

Conditional Evaluators determine when events should trigger and also when subscribers should be notify to execute adaptation. AdaptiFlow implicitly provides two evaluator types: (1) **Threshold-Based Evaluators**, simple rules like *GreaterThan* or *Between*, ideal for tests on numerical data types (e.g., scaling when CPU > 85%) and (2) **Context-Aware Evaluators**, custom logic combining multiple metrics. For instance, a *PeakHourEvaluator* might disable non-essential features during high traffic only if cloud credits are low. Conditional Evaluators acts like filters to know when adaptation actions will be executed.

3.5 Event Specification

Events in AdaptiFlow are defined as logical combinations of conditions evaluated against collected metrics. The *Event Specification* component allows developers to declaratively construct events using threshold-based or custom logic. Events serve as the bridge between raw metrics and actionable insights. here is some examples of events: (1) **Threshold-Based Events**, simple rules like *HighCPUTEvent* (CPU > 80%) or *LowDiskSpaceEvent* (FreeDisk < 10%) and (2) **Custom Events**, multi-condition rules such as `ServiceDegradationEvent` (latency > 1s && error rate > 10%).

3.6 Event Observation

The *Event Observation* component implements strategies to detect specified events, balancing timeliness and resource efficiency: (1) **Periodic Polling**, checks conditions at fixed intervals (e.g., CPU every 30 seconds) for gradual trends like memory leaks and (2) **On-Demand Triggers**, event-driven checks (e.g., during API failures) for rapid response to anomalies. The `Observation Scheduler` orchestrates these strategies and developers can customize polling intervals or define their own observation logic, ensuring flexibility across scenarios.

3.7 Event Subscription

AdaptiFlow's *Event Subscription* model allow services or service components (parts of the service) to declaratively register interest in specific events. When an event triggers, the notification of subscribers consist of the execution of their adaptation actions. The adaptation action can be a local action inside the current service (e.g. EnableCache, LowPowerMode), an API call to another service to execute some adaptations actions remotely (e.g. OpenCirCuitBreaker) or the specification of another adaptation scenario (e.g. DDoS Attack Mitigation).

This model supports hybrid architectures: a service can act as both a subscriber (e.g., Auth service responding to `DDoSAttackEvent`) and an event emitter (e.g., emitting `HighLatencyEvent`). Subscriptions can be dynamically updatable, allowing runtime adjustments without service restarts.

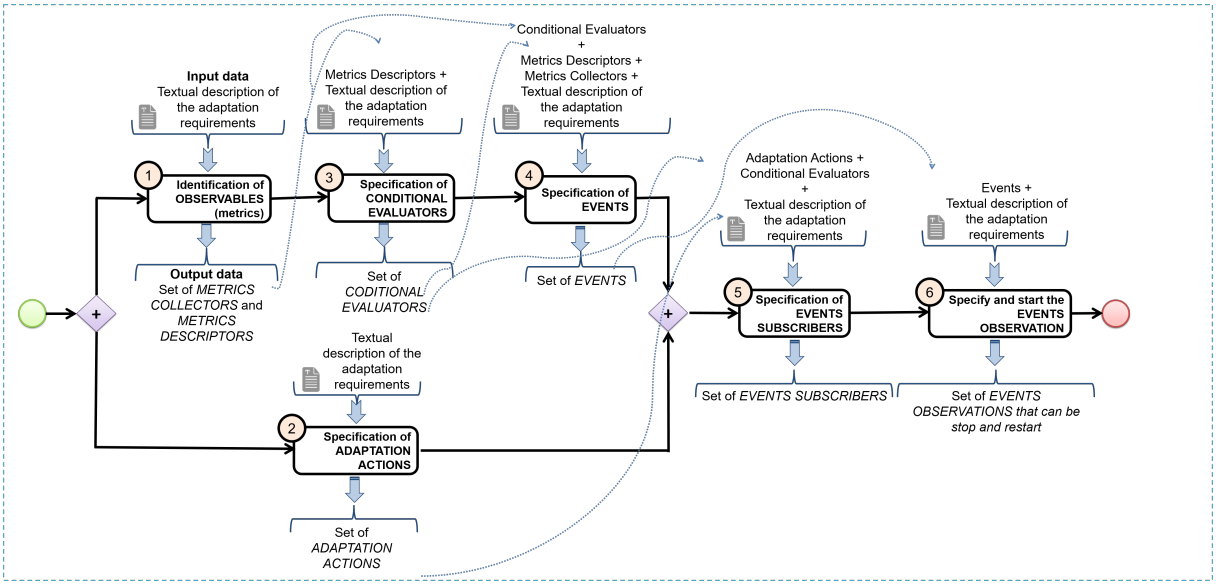

Figure 3: AdaptiFlow Workflow to enable adaptability in a given cloud microservices system using the textual description of the adaptation requirements or adaptation scenario description.

The preceding sections detailed AdaptiFlow's core components, which collectively enable decentralized, context-aware adaptations. However, realizing these capabilities requires a systematic methodology to translate high-level adaptation requirements (e.g., "prevent service outages during traffic spikes") into concrete implementations. This methodology—the **Workflow for Enabling Adaptability**—guides developers through six stages, each leveraging AdaptiFlow's components to incrementally build self-adaptive logic:

- **Metrics Collectors** operationalize the *Identify Observables* stage, defining what data to gather.
- **Adaptation Actions** and **Event Subscription** drive the *Specify Event Subscribers* stage, linking events to executable responses.
- **Conditional Evaluators** and **Event Specification** underpin the *Specify Conditional Evaluators* and *Specify Events* stages, mapping raw data to actionable triggers.
- **Event Observation** implements the *Configure Event Observation* stage, ensuring timely detection.

The following subsection elucidates this workflow, demonstrating how developers progress from abstract requirements (e.g., mitigating DDoS attacks) to deployable adaptation logic.

3.8 Workflow for Enabling Adaptability with AdaptiFlow

AdaptiFlow's workflow (Fig. 3) transforms textual adaptation requirements into executable adaptation logic through six systematic steps. To illustrate this process, consider a scenario where the TeaStore Persistence service [18] detects database timeouts and coordinates graceful degradation across dependent services (Auth, Recommender, Image, WebUI). The workflow begins with the identification of critical data points and culminates in the implementation of observation strategies, ensuring end-to-end adaptability.

Step 1: Identify Observables The process starts by defining *observables*—data sources required to monitor the system's state. Developers parse adaptation requirements to determine which metrics (e.g., database health, API latency) must be collected. For the Adaptable TeaStore scenario [2], this involves creating `DatabaseHealthCollector` to track connection timeouts. AdaptiFlow provides abstract interfaces (e.g., `IMetricsCollector`) to model these observables, decoupling data collection from downstream logic. Metrics descriptors define the structure of collected data (e.g., sampling frequency, data type), enabling consistent interpretation across services.

Step 2: Define Adaptation Actions (Parallel Step) Concurrently, developers specify *adaptation actions*—concrete steps to achieve adaptation goals. These actions fall into two categories: *infrastructure-level* (e.g., restarting pods, scaling instances) and *business-level* (e.g., disabling recommendations, enabling maintenance modes). For Adaptable TeaStore, the Persistence service defines `EnableCache`, while the WebUI implements `EnableMaintenanceMode` and the recommender implements `LowPowerMode`. AdaptiFlow abstracts action execution through interfaces (e.g., `IAdaptationAction`), allowing delegation to external tools (e.g., Docker API, Kubernetes API) or custom logic. This separation ensures developers focus on defining *what* to adapt, not *how* to implement low-level operations.

Steps 1 and 2 constitute the *preparation phase*, equipping services with the interfaces required for autonomic management. By defining metrics collectors (Step 1), services expose monitoring endpoints that provide real-time insights into their state (e.g., database health, API latency). Simultaneously, specifying adaptation actions (Step 2) establishes control points—actuators that enable runtime adjustments (e.g., restarting pods, toggling features). This aligns with the Autonomic Computing Vision (ACV), where "the managed element is adapted to enable the autonomic manager to monitor and control it" [7]. AdaptiFlow operationalizes this by decoupling data collection and action execution into reusable interfaces (`IMetricsCollector`, `IAdaptationAction`), effectively transforming non-adaptive services into autonomic elements. Once prepared, services offer standardized APIs for observation and control, enabling the autonomic manager (or decentralized logic) to implement adaptation scenarios without invasive code changes.

Step 3: Specify Conditional Evaluators Conditional evaluators encode the logic for triggering events and filtering subscribers. This step involves dual evaluations: (1) *event evaluators* determine if an event should trigger (e.g., `DatabaseTimeoutEvaluator` checks for consecutive timeouts), and (2) *subscriber evaluators* decide which subscribers should act (e.g., Auth service acts only after a 5-minute outage). Evaluators leverage collected metrics and can integrate external APIs (e.g., machine learning models for anomaly detection). For our Adaptable TeaStore, a `UnHealthyDatabaseEvaluator` combines database health metrics with service dependency statuses to assess system stability.

Step 4: Specify Events Events semantically encapsulate adaptation scenarios. Developers bind evaluators and metrics to named events (e.g., `DatabaseUnavailableEvent`), which act as triggers for non coordinated actions. Events inherit from AdaptiFlow's `ConditionalEvent` base class, enabling reuse across scenarios. For Adaptable TeaStore, the `DatabaseUnavailableEvent` is defined using the `LocalDatabaseMetricsCollector` and `UnHealthyDatabaseEvaluator`, ensuring it triggers only when timeout thresholds are breached. Events provide human-readable context (e.g., "database_unavailable"), aligning with adaptation goals described in requirements.

Step 5: Specify Event Subscribers Subscribers declaratively register for events and define action execution strategies. A subscriber comprises (1) a list of adaptation actions and (2) a conditional evaluator to filter notifications. In Adaptable TeaStore, the Persistence service subscribes to `DatabaseUnavailableEvent` with a `EnableCache` action, while the WebUI service switches to maintenance mode. Subscribers can be granular (e.g., specific UI components) or service-wide. AdaptiFlow supports strategies like *immediate execution* (act on first trigger) or *event counting* (act after N occur-

rences), offering flexibility akin to MAPE-K's planning phase without fully implement a planner.

Step 6: Configure Event Observation The final step defines *how* events are detected. AdaptiFlow supports periodic polling (e.g., check database health every 10s) or on-demand triggers (e.g., during API failures). For Adaptable TeaStore, the `ObservationScheduler` uses periodic checks for database health. Developers can extend the `AbstractObservationScheduler` class to implement custom strategies (e.g., event-driven checks via message queues), ensuring adaptability to platform constraints. Observation configurations are decoupled from event logic, allowing runtime adjustments without disrupting active adaptations.

This structured workflow ensures systematic implementation of adaptation scenarios while preserving microservices' autonomy. Section 4 validates AdaptiFlow's efficacy through three scenarios in the Adaptable TeaStore: *self-healing* (database recovery), *self-protection* (DDoS mitigation), and *self-optimization* (traffic management).

4 Case Study: Building the Adaptable TeaStore

Validation of AdaptiFlow's design principles was carried out through comprehensive experiments with the TeaStore benchmark in order to provide an adaptable version: Adaptable TeaStore previously described by Bliudze et al [2]. This case study has two main objectives: (1) to demonstrate the practical implementation of our abstraction layers and (2) to evaluate the effectiveness of the framework through distinct adaptation scenarios. We chose TeaStore [18] for its representative cloud-native architecture comprising five interdependent services (Auth, Persistence, Recommender, Image and WebUI). The experimental methodology systematically examines three autonomous capabilities by implementing three adaptation scenarios: self-healing (service recovery), self-protection (DDoS mitigation), and self-optimization (traffic management).

4.1 Experimental Setup

The experiments were conducted on a Docker-based [10] environment with Portainer CE [1] for container management. Each TeaStore service (Auth, Persistence, Recommender, Image, WebUI) was instrumented with AdaptiFlow's abstraction layer, exposing standardized REST APIs and java classes for metrics collection and adaptation action execution. The AdaptiFlow framework library (compiled with JDK 11) provides base classes and interfaces for implementing metrics collectors, adaptation actions and event handlers. The table (Tab. 1) summarize the experimental setup.

We utilized the HTTP load generator [18] with Limbo [8] for modeling load intensities just like it is done in the original TeaStore. We containerized the two components of the load generator (the director and the load generator) for Docker compatibility. Load intensity was controlled via three CSV profiles: (1) *increasingLowIntensity.csv* for gradual ramp-up, (2) *increasingMedIntensity.csv* for moderate ramp-up and (3) *increasingHighIntensity.csv* for aggressive ramp-up.

To simplify things, we use the *increasingHighIntensity* profile to simulated DDoS attack conditions, while *increasingMedIntensity* tested self-optimization thresholds. Locust was present in the original configuration, but we intentionally ignored it as the limbo http load generator was sufficient for our experiments.

Each service's Docker container included: (1) *Metrics Collectors* (Infrastructure / Business-level), (2) *Adaptation Actions* (Business-level only) and *Event Handlers*. As our main objective was to define the abstraction layers needed to specify metrics collectors and adaptation actions, we have not addressed

the implementation details of adaptation actions at the infrastructure level (e.g. stopping or restarting containers), since applications such as portainer [1] demonstrate the feasibility of such actions in docker and kuberbates environments. We focused more on implementing adaptation actions linked to the business logic of the various microservices (e.g. optimizing recommendation, enabling/disabling caching, using an external provider for images, etc.). In addition, the validation focused on Docker; Kubernetes behavior was verified through API responses but was not tested in cluster orchestration scenarios.

This setup enabled systematic evaluation of AdaptiFlow's ability to translate adaptation requirements into runtime behavior adjustments. The following subsections detail our implementation of three autonomic scenarios, demonstrating how AdaptiFlow's abstraction layer bridges the gap between non-adaptive services and self-managing elements.

4.2 Self-Healing: Database Unavailability

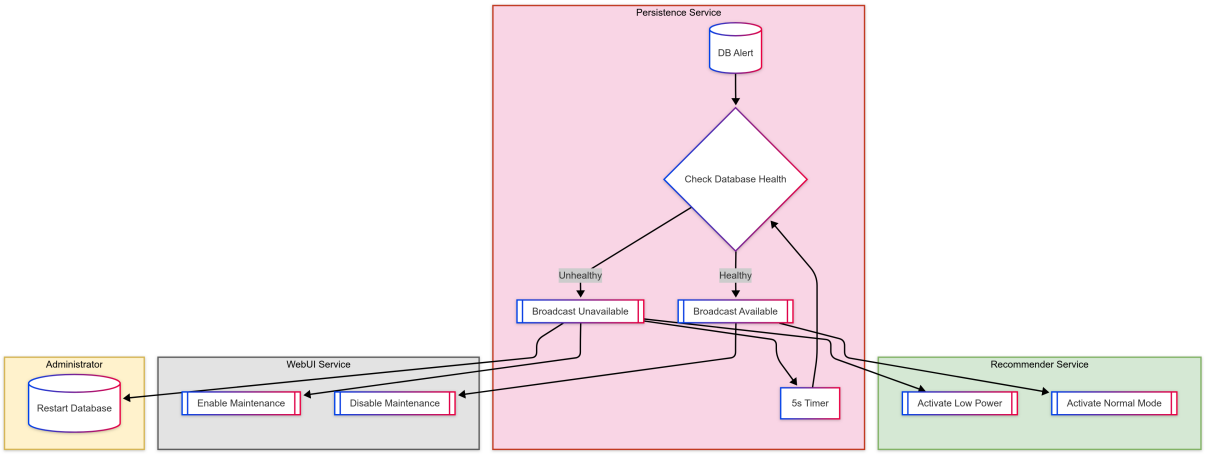

Figure 4: The database unavailable mitigation flow diagram.

Scenario overview Figure 8: The system is deployed in a barebone configuration with local services (Auth, Recommender, Image, Persistence, WebUI). The Persistence service detects timeouts from the local database due to an unexpected interruption. It triggers adaptation actions across dependent services (Auth, Recommender, Image, WebUI) to gracefully degrade functionality. The WebUI displays a maintenance message, and the system administrator is alerted to restart the database. Once restored, services resume normal operation.

In this scenario, the Persistence microservice serves as the central adaptation coordinator, monitoring database health through three key metrics: *connection status* (boolean), *query response times* (milliseconds), and *active connection counts*. Upon detecting failures, it triggers cascading adaptations across dependent components. The WebUI responds by displaying maintenance pages, while the Recommender service dynamically adjusts its algorithm between normal operation (popular items only) and low-power mode (no recommendations).

Metrics Collection: The Persistence service's *LocalDatabaseMetricsCollector* aggregates four critical database health indicators: *response times*, *network status*, *active connections*, and *pending queries*. Collection occurs through a combination of periodic polling define by the observation strategy (5-second intervals), JDBC health checks with configurable timeouts, and connection pool monitoring. This multi-metric approach ensures reliable failure detection while minimizing false positives.

Action Execution: Four adaptation patterns coordinate the system response. *Event broadcasting* propagates status changes via REST notifications, while cache management dynamically *enables/disables* caching to improve system fault tolerance. The UI degradation pattern activates maintenance displays, and service throttling reduces computational load through the Recommender's power modes. These actions demonstrate AdaptiFlow's ability to easily combine infrastructure and business-level adaptations.

Adaptation Logic: The system employs two condition evaluators with distinct triggering mechanisms. The *UnHealthyDatabaseEvaluator* activates when response times exceed 5000ms or when the network status deviates from expectations. Conversely, the *HealthyDatabaseEvaluator* requires both sub-5000ms response times and proper network status before signaling recovery. The architectural importance stems from the persistence service's dual role as managed resource and adaptation orchestrator. Bearing in mind that, once the metrics collectors and adaptation actions have been defined, it is entirely possible to define an autonomous manager (or adaptation logic), either exogenous (an agent, for example) or internal (adaptation rules), as illustrated in the following example.

Implementation Notes: Current limitations include REST-based event propagation (planned upgrade to message brokers) and manual database recovery (future automation target). These constraints were intentionally maintained during validation to isolate and demonstrate the framework's core capabilities. The complete Java implementation, including metrics collectors, evaluators, and action handlers, appears in Appendix D.2.

4.3 Self-Protection: Mitigating DDoS Attacks

Scenario overview Figure 9: The system responds to a *Distributed Denial-of-Service* attack through a layered verification and adaptation protocol. The WebUI service acts as the primary detector, analyzing traffic patterns through requests rate monitoring before declaring an attack only after three consecutive verification stages. Upon confirmation, it simultaneously activates local circuit breaker and broadcasts REST notifications to all dependent services. Each recipient service—Auth, Recommender, Persistence, and Image—performs its own dual-layer validation of attack indicators before implementing specialized protections. First of all, the entire services activates their local circuit breaker then the Recommender downgrades to low-power operation mode. The Persistence service shifts to cache-only operations, and the Image service reroutes requests to external providers.

This scenario describe a multi-stage approach that combines centralized detection with decentralized execution, ensuring attack mitigation while preserving critical functionality through coordinated circuit breaker deployment and service-specific degradation modes. The entire process emphasizes verification rigor, requiring five total confirmation checks (three central and two local) before full protective measures engage, preventing false positives during normal traffic spikes.

Metrics Collection: In this scenario, we employs uniform metric collection across all services through *LocalRequestMetricsCollectors*, monitoring *request rates*, *IP patterns*, *error frequencies* and *requests details*. The WebUI service initiates detection using threshold-based evaluators (*DDoSEvaluator* / *NonDDoSEvaluator*) that analyze requests per second over 60-second windows: more complex verifications could have been done but we decide to simplify things for our experiments. While all services can track contextual traffic metrics, at the beginning of the process, only WebUI detect and triggers system-wide adaptations when its evaluator detects sustained rates exceeding 300 requests/second. Then WebUI notify all services to start their own decentralized monitoring and detection.

Action Execution: Adaptations cascade through a two-phase protocol. The WebUI service first activates maintenance pages and circuit breaker after triple-confirmed detection, then broadcasts alerts

via REST. Recipient services implement specialized protections: Auth activates circuit breaker, Recommender switches to low power mode (no recommendations), Persistence prioritizes cached data, and Image services reroute to external providers. Each service independently verifies attack conditions through dual local checks before executing actions preventing single-point failures.

Adaptation Logic: The system employs layered verification to balance responsiveness with reliability. WebUI requires three consecutive metric evaluations exceeding the threshold to declare an attack, reducing false positives during traffic spikes. Downstream services perform two local confirmations using their own metrics before activating protections. This hierarchical logic combines global attack awareness with local context validation, allowing services to autonomously scale their response intensity—from full circuit breaking in Auth to graceful degradation in Recommender.

Implementation Notes: The event-driven architecture uses *ConditionalEvent* wrappers around metric streams, with WebUI's *EventCounterSubscriber* requiring three threshold breaches before triggering adaptations. Recipient services reuse the same evaluator classes but configure two-step verification. REST notifications propagate through standardized adaptation endpoints, enabling heterogeneous action execution—from UI changes (*EnableMaintenanceMode*) to infrastructure adjustments (*EnableExternalImageProvider*). The code structure demonstrates AdaptiFlow's template pattern, where services implement shared interfaces (*IAdaptationAction*, *ConditionEvaluator*) while customizing verification thresholds and action combinations.

4.4 Self-Optimization: Handling Benign Traffic Surges

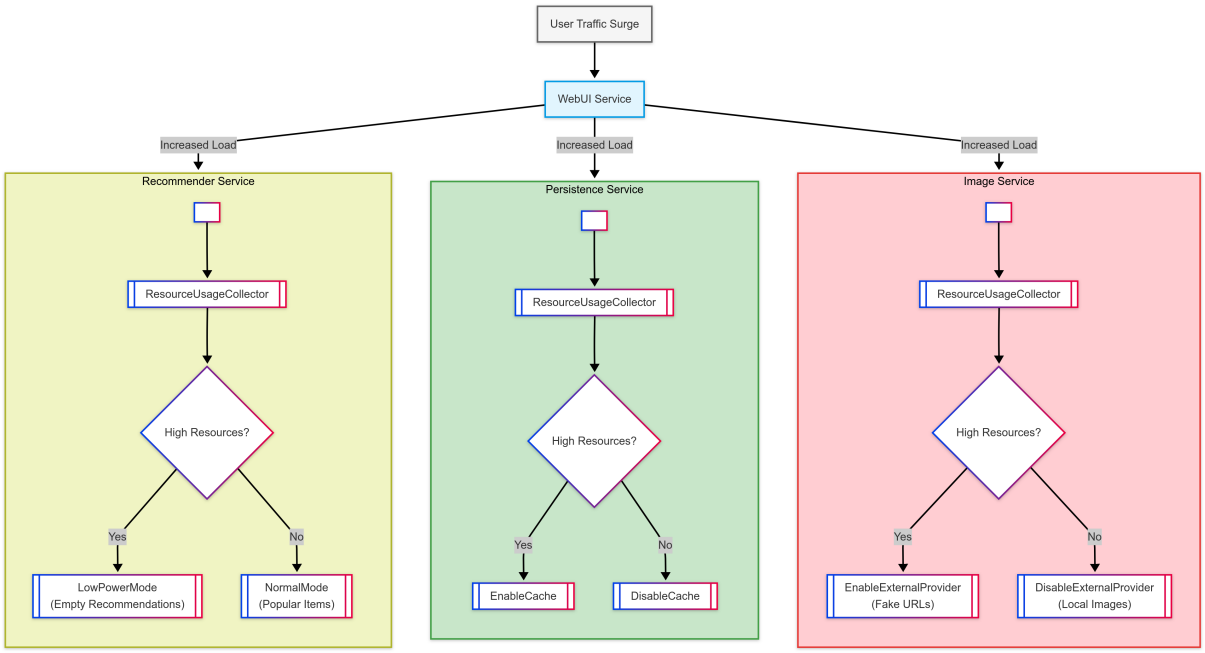

Figure 5: Benin traffic adaptation workflow

Scenario overview Figure 5: The system autonomously adapts to legitimate traffic spikes through context-aware resource optimization when infrastructure scaling is unavailable. Faced with surging user demand, the WebUI service triggers decentralized adaptations across dependent components without

centralized coordination. The Recommender service switches to no recommendations, reducing computational load. Simultaneously, the Persistence service activates caching mechanisms, prioritizing frequently accessed data to alleviate database pressure. Image processing workloads shift dynamically to external provider. Each service independently monitors resource utilization through dedicated collectors—CPU and memory-to calibrate adaptations, enabling system-wide load balancing.

In this scenario, we use an uncoordinated optimization strategy that demonstrates AdaptiFlow's ability, where autonomous decisions at service level collectively contribute to stabilizing the system despite resource constraints. This stabilization is sometimes achieved at the expense of the user experience, depending on the adaptation actions chosen.

Metrics Collection: we employ decentralized resource monitoring through service-specific *ResourceUsageCollectors*, tracking CPU and memory utilization. Each service (Recommender, Persistence, Image) independently observes metrics via hash maps containing numeric percentages for *cpu_usage* and *memory_usage*. Threshold-based evaluators trigger adaptations when CPU exceeds 75% or memory surpasses 80%, with recovery initiated when both metrics drop below 60%. This dual-threshold approach prevents oscillations during borderline resource conditions while maintaining granular control over optimization triggers.

Action Execution: Services implement autonomous optimization strategies when thresholds breach. The Recommender reduces computational load by switching to no recommendations via *LowPowerMode*, cutting recommendation logic overhead. Persistence activates caching mechanisms (*EnableCache*) to minimize database queries, while the Image service offloads processing to external provider through *EnableExternalImageProvider*. These localized adaptations occur without inter-service coordination, allowing parallel optimization while preserving core functionality. Recovery actions revert systems to normal operations when resource usage stabilizes.

Adaptation Logic: Decentralized decision-making leverages identical threshold patterns across services with service-specific interpretations. The *IncreaseResourceUsageEvaluator* class triggers optimizations when either CPU or memory exceeds upper limits, while *DecreaseResourceUsageEvaluator* restores normal operations when both metrics fall below recovery thresholds. This asymmetric logic prioritizes rapid response to overloads while requiring sustained improvement for recovery, balancing stability with responsiveness. Services maintain isolation, Rescommender never influences Persistence caching decisions.

Implementation Notes: The architecture uses standardized *ConditionalEvent* wrappers around resource metrics, enabling code reuse across services while allowing threshold customization. HashMaps structure metrics with strict key conventions (*cpu_usage*, *memory_usage*) for evaluator compatibility. Five-second polling intervals (configurable via *EVENT_LISTENING_INTERVAL_MS*) balance detection speed with overhead. Though services share adaptation patterns, each can maintains separate threshold configurations.

5 Conclusion and future work

The development and evaluation of AdaptiFlow demonstrate a significant advance in self-adaptation within cloud-native microservices. This section consolidates the framework's main contributions and empirical validation, while outlining ways to extend its capabilities in both theoretical and practical dimensions.

5.1 Summary

AdaptiFlow addresses the critical challenge of enabling self-adaptation in cloud-native microservices through an abstraction layer focused on the **Monitor** and **Execute** phases of the MAPE-K loop. By decoupling metrics collection and action execution from adaptation logic, the framework provides standardized interfaces that transform conventional services into autonomic elements without invasive code changes. While our primary goal centered on monitoring / execution abstractions, we introduced lightweight rule-based mechanisms for the **Analyze** and **Plan** phases to validate the core architecture through realistic scenarios. This pragmatic approach demonstrates how decentralized, event-driven adaptations can emerge from localized decisions while maintaining system-wide coherence.

The framework's strength lies in its dual-purpose API design: developers instrument services with metrics collectors and actuators using unified interfaces, while adaptation strategists (human operators or AI agents) leverage these primitives to implement diverse control policies. Our validation through the **Adaptable TeaStore**: a variant of the original TeaStore that we built by enriching it with autonomous capabilities and comprehensive documentation confirms AdaptiFlow's practicality across three critical adaptation objectives: self-healing, self-protection, and self-optimization.

5.2 Future Directions

Ongoing work focuses on three key enhancements:

- **Formal Coordination**: Integration with *JavaBIP* [3] to manage conflicting adaptations through rigorous behavioral models, preventing undesirable interactions between concurrent strategies.
- **Adaptation Language**: Exploration of a domain-specific language (DSL) for declarative rule or adaptation scenario specification, lowering the barrier for non-programmers to define adaptation policies.
- **Intelligent Adaptation**: Implementation and evaluation of AI-driven strategies using the AWARE framework [16], comparing machine learning approaches with traditional rule-based methods in terms of responsiveness and resource efficiency.

The instrumented **Adaptable TeaStore** provides an ideal testbed for these advancements, offering preconfigured metrics and actuators for benchmarking adaptation techniques. Future studies will quantify the performance trade-offs between centralized vs. decentralized decision-making and assess the framework's scalability in large-scale deployments.

By bridging the gap between autonomic computing theory and cloud computing practice, AdaptiFlow lays the foundations for next-generation self-adaptive systems, in which microservices autonomously navigate dynamic environments, while developers retain full (ideal case) or moderate (if necessary) control over adaptation semantics. This balance between automation and flexibility makes the framework an essential tool for resilient, efficient cloud architectures at a time of ever-increasing operational complexity.

References

[1] *Welcome | Portainer Documentation — docs.portainer.io*. https://docs.portainer.io/. [Accessed 20-04-2025].

[2] Simon Bliudze, Giuseppe De Palma, Saverio Giallorenzo, Ivan Lanese, Gianluigi Zavattaro & Brice Arleon Zemtsop Ndadji (2024): *Adaptable TeaStore*. arXiv preprint arXiv:2412.16060.

[3] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek & Alina Zolotukhina (2017): *Exogenous coordination of concurrent software components with JavaBIP*. Software: Practice and Experience 47(11), pp. 1801–1836, doi:10.1002/spe.2495.

[4] Sharmin Jahan, Ian Riley, Charles Walter, Rose F Gamble, Matt Pasco, Philip K McKinley & Betty HC Cheng (2020): *MAPE-K/MAPE-SAC: An interaction framework for adaptive systems with security assurance cases*. Future Generation Computer Systems 109, pp. 197–209.

[5] Saeid Jamshidi, Ashkan Amirnia, Amin Nikanjam & Foutse Khomh (2024): *Enhancing security and energy efficiency of cyber-physical systems using deep reinforcement learning*. Procedia Computer Science 238, pp. 1074–1079.

[6] Joao Paulo Karol Santos Nunes, Shiva Nejati, Mehrdad Sabetzadeh & Elisa Yumi Nakagawa (2024): *Self-adaptive, requirements-driven autoscaling of microservices*. In: Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 168–174.

[7] Jeffrey O Kephart & David M Chess (2003): *The vision of autonomic computing*. Computer 36(1), pp. 41–50.

[8] Jóakim von Kistowski, Nikolas Roman Herbst & Samuel Kounev (2014): *LIMBO: A Tool For Modeling Variable Load Intensities*. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), ICPE '14, ACM, New York, NY, USA, p. 225–226.

[9] Michael Maurer, Ivan Breskovic, Vincent C Emeakaroha & Ivona Brandic (2011): *Revealing the MAPE loop for the autonomic management of cloud infrastructures*. In: 2011 IEEE symposium on computers and communications (ISCC), IEEE, pp. 147–152.

[10] Dirk Merkel (2014): *Docker: lightweight linux containers for consistent development and deployment*. Linux journal 2014(239), p. 2.

[11] Andreas Metzger, Clément Quinton, Zoltán Ádám Mann, Luciano Baresi & Klaus Pohl (2024): *Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration*. Computing 106(4), pp. 1251–1272.

[12] Jiyoung Oh, Claudia Raibulet & Joran Leest (2022): *Analysis of MAPE-K loop in self-adaptive systems for cloud, IoT and CPS*. In: International Conference on Service-Oriented Computing, Springer, pp. 130–141.

[13] Michael Riegler, Johannes Sametinger & Michael Vierhauser (2023): *A distributed MAPE-K framework for self-protective IoT devices*. In: 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE, pp. 202–208.

[14] Stuart J Russell & Peter Norvig (2016): *Artificial intelligence: a modern approach*. pearson.

[15] Mauricio Salatino, Mariano De Maio & Esteban Aliverti (2016): *Mastering jboss drools 6*. Packt Publishing Ltd.

[16] Brell Peclard Sanwouo, Paul Temple & Clément Quinton (2025): *Breaking the Loop: AWARE is the new MAPE-K*. In: FSE'25-International Conference on the Foundations of Software Engineering.

[17] Marco Stadler, Johannes Sametinger & Michael Riegler (2024): *Cyber-resilient edge computing: a holistic approach with multi-level MAPE-K loops*. In: 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), IEEE, pp. 79–83.

[18] Joakim Von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann & Samuel Kounev (2018): *Teastore: A micro-service reference application for benchmarking, modeling and resource management research*. In: 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, pp. 223–236.

[19] Fenglin Yu, Fangkai Yang, Xiaoting Qin, Zhiyang Zhang, Jue Zhang, Qingwei Lin, Hongyu Zhang, Yingnong Dang, Saravan Rajmohan, Dongmei Zhang et al. (2025): *Enabling Autonomic Microservice Management through Self-Learning Agents*. arXiv preprint arXiv:2501.19056.

[20] Zhiyang Zhang, Fangkai Yang, Xiaoting Qin, Jue Zhang, Qingwei Lin, Gong Cheng, Dongmei Zhang, Saravan Rajmohan & Qi Zhang (2024): *The Vision of Autonomic Computing: Can LLMs Make It a Reality?* arXiv preprint arXiv:2407.14402.

Appendices

A Adaptable TeaStore and AdaptiFlow essential links

- Adaptable TeaStore repository with full documentation: https://gitlab.inria.fr/adaptable-teastore/experimentation-platform.
- Adaptiflow documentation: click here
- Adaptation Scenario Overview: https://gitlab.inria.fr/adaptable-teastore/experimentation-platform/-/blob/main/Sources/utilities/tools.spirals.adaptableteastore.adaptiflow/SCENARIOS_OVERVIEW.md
- Adaptable TeaStore docker containers: https://hub.docker.com/repositories/cerberus237

B AdaptiFlow Class diagram

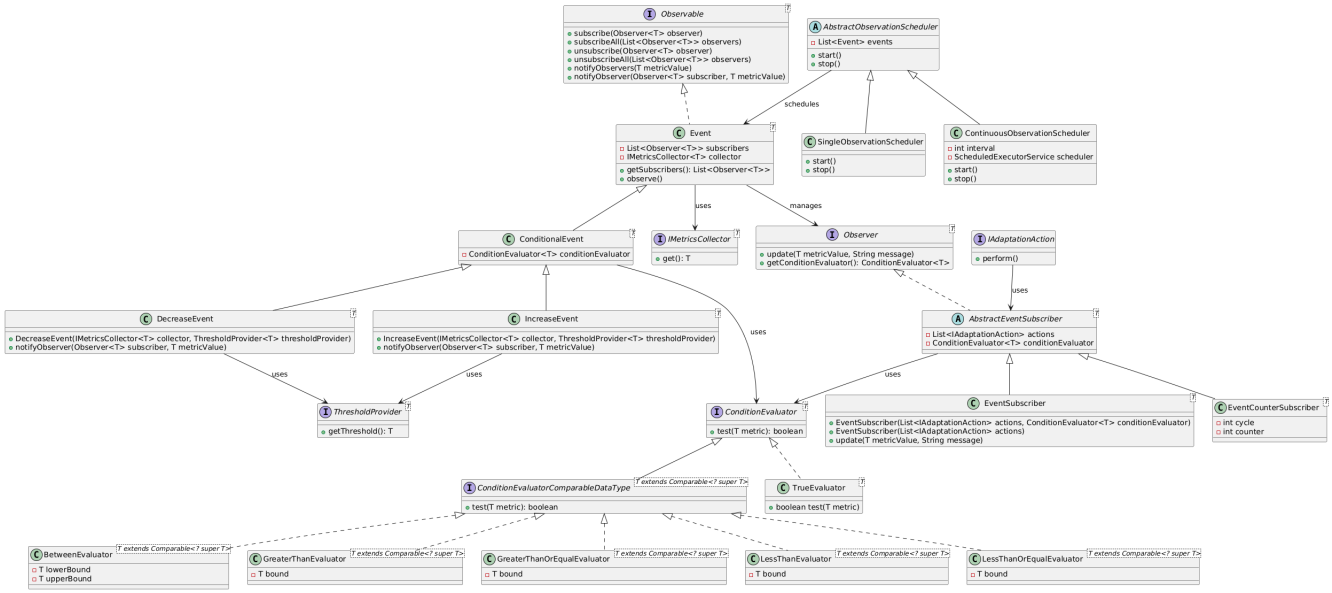

Figure 6: AdaptiFlow Class diagram.

Figure 6 presents the structural design of AdaptiFlow through its key components and their relationships. The architecture follows a modular design pattern that separates concerns between metric collection, event processing, and adaptation execution.

B.1 Key Interfaces

- `IMetricsCollector<T>`
  - **Purpose**: Standardizes metric collection across different data sources
  - **Method**: `T get()` - Retrieves current metric values
- `ThresholdProvider<T>`

  - **Purpose**: Abstracts threshold configuration for dynamic adaptation
  - **Method**: `T getThreshold()` - Supplies comparison boundaries

B.2 Core Classes

- `Event<T>` (Abstract Base Class)
  - **Responsibilities**:
    * Manages observer subscriptions via `subscribe(Observer<T>)`
    * Coordinates notifications through `notifyObservers(T metricValue)`
  - **Concrete Implementations**:
    * `IncreaseEvent`: Triggers when metrics exceed upper bounds
    * `DecreaseEvent`: Activates when metrics fall below thresholds
- `ConditionEvaluator<T>` (Abstract)
  - **Method**: `boolean test(T metric)` - Evaluation contract
  - **Implementations**:
    * `GreaterThanEvaluator`: Checks metric > threshold
    * `LessThanEvaluator`: Verifies metric < threshold
- `ObservationScheduler`
  - **Variants**:
    * `SingleObservationScheduler`: One-time evaluation
    * `ContinuousObservationScheduler`: Periodic monitoring
- `EventSubscriber<T>`
  - **Fields**:
    * `List<IAdaptationAction> actions`: Executable responses
    * `ConditionEvaluator<T> evaluator`: Trigger conditions

B.3 Architectural Relationships

- **Usage Dependencies**:
  - `Event` aggregates `IMetricsCollector` for data access
  - Evaluators consume `ThresholdProvider` for dynamic boundaries
- **Inheritance Hierarchy**:
  - Specialized evaluators extend `ConditionEvaluator`
  - Event variants inherit from base `Event` class
- **Composition**:
  - `EventSubscriber` contains collections of `IAdaptationAction`
  - `ObservationScheduler` manages `Event` instances

B.4 Implementation Example

```
public class GreaterThanEvaluator<T extends Comparable<? super T>>
implements ConditionEvaluatorComparableDataType<T> {

    // The value that the metric will be compared against
    private final T bound;

    // Constructor to initialize the evaluator with the comparison bound
    public GreaterThanEvaluator(T bound) {
        this.bound = bound;
    }

    /**
     * Evaluates whether the given metric is greater than the defined bound.
     *
     * @param metric the metric to evaluate
     * @return {@code true} if the metric is greater than the bound; {@code
false} otherwise
     */
    @Override
    public boolean test(T metric) {
        // Compare the metric with the bound and return true if metric is
greater
        return metric.compareTo(bound) > 0;
    }
}

```

Listing 1: Threshold Evaluation Implementation

The class diagram demonstrates AdaptiFlow's extensible design, where new metric sources, evaluation strategies, and adaptation actions can be integrated without modifying core components. This flexibility supports both infrastructure-level adaptations (e.g., scaling) and business logic adjustments (e.g., feature toggles) through a consistent interface pattern.

C Sequence Diagram

The sequence diagram in Figure 7 details the runtime interaction workflow between AdaptiFlow's core components. The process flow comprises five key phases:

1. Observation Scheduler Initiation The adaptation cycle begins when the `ObservationScheduler` use the defined observation strategy to observe events. This component can act as the system's heartbeat, initiating each adaptation cycle at configurable intervals (default: 5 seconds).

2. Metrics Collection Upon scheduler activation, the `MetricsCollector` gathers current system state data, including both infrastructure metrics (CPU, memory) and business-level indicators (request rates, error counts). Collected metrics are packaged into a standardized format and transmitted to the `EventManager`.

3. Condition Evaluation The `EventManager` processes incoming metrics through registered `ConditionEvaluator` instances. Each evaluator applies threshold-based logic (e.g. "CPU > 80%") or custom business rules to determine whether events and then adaptation actions should be triggered..

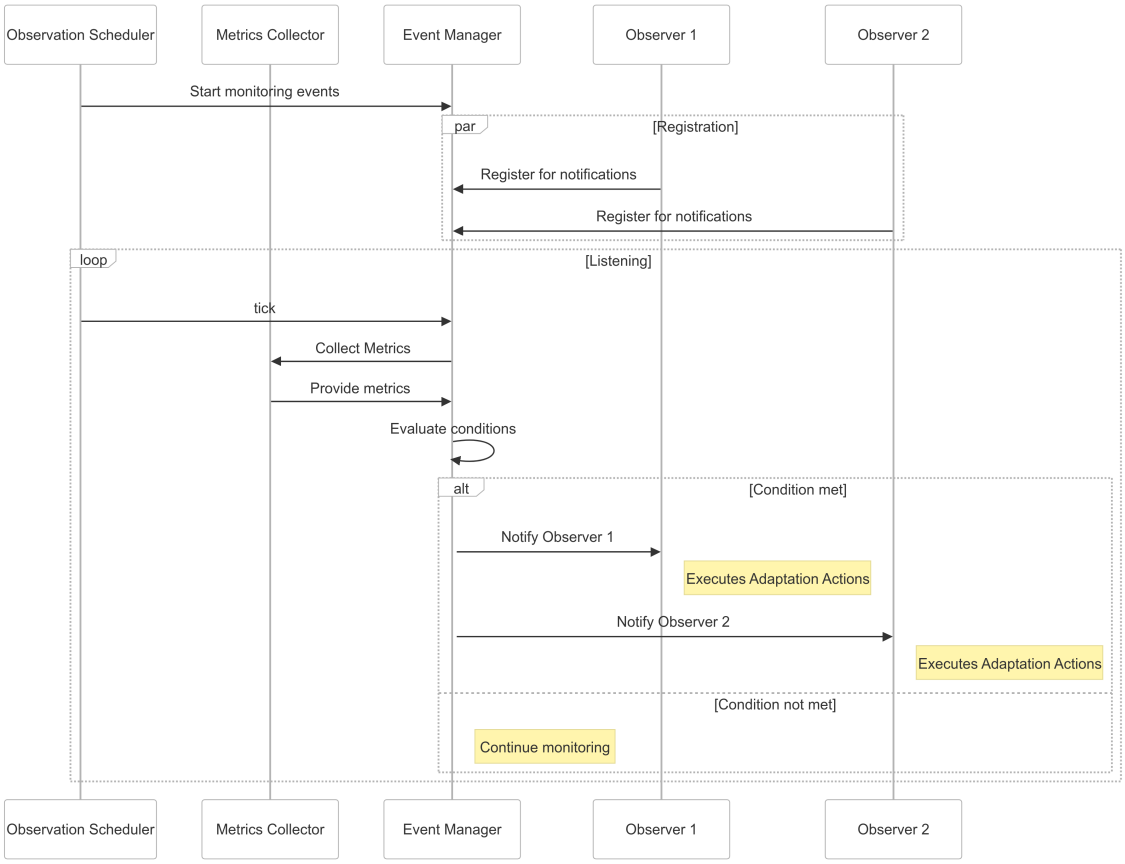

Figure 7: Sequence diagram illustrating runtime interactions between AdaptiFlow components

4. Event Notification When conditions are met, the `EventManager` notifies all subscribed components through the publish-subscribe pattern. Notifications include both the triggered event type (e.g., `HighCPUTEvent`) and relevant metric snapshots for contextual adaptation decisions.

5. Action Execution Subscribed services execute their registered `IAdaptationAction` implementations.

This sequence demonstrates AdaptiFlow's ability to coordinate decentralized adaptations while maintaining loose coupling between components. The clear separation between monitoring, evaluation, and execution phases enables flexible extension of individual components without system-wide modifications.

D Database Unavailable Scenario Implementation

D.1 Code Implementation

```
1        public class HealthyDatabaseEvaluator implements ConditionEvaluator<
   SQLDatabaseMetrics> {
2            // Maximum acceptable response time for the database in milliseconds
3            private final Long maxResponseTime;
4
```

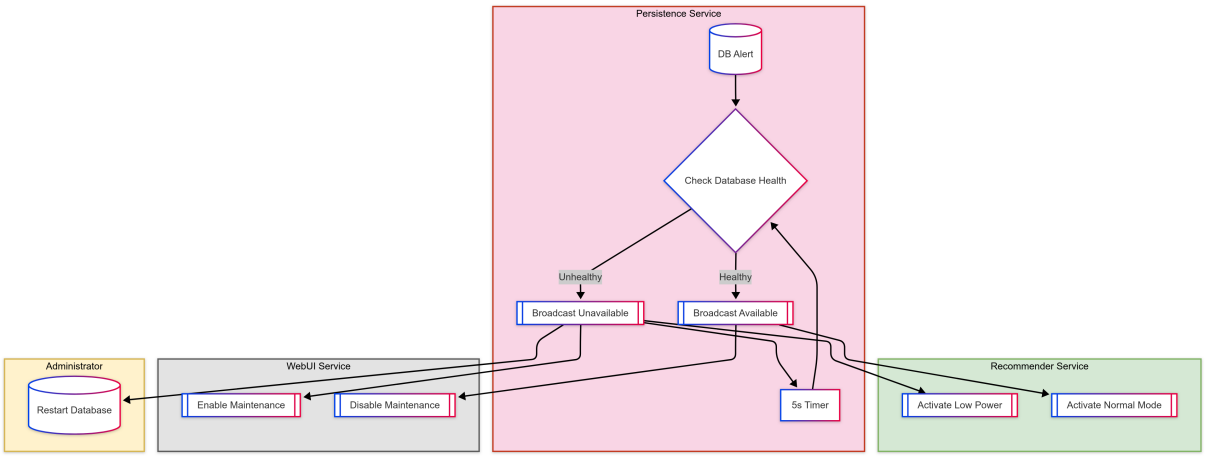

Figure 8: The database unavailable mitigation flow diagram.

```
        // Expected network status (true for healthy, false for unhealthy)
        private final boolean expectedNetworkStatus;

        // Constructor to initialize the evaluator with specified conditions
        public HealthyDatabaseEvaluator(Long maxResponseTime, boolean
expectedNetworkStatus) {
            this.maxResponseTime = maxResponseTime;
            this.expectedNetworkStatus = expectedNetworkStatus;
        }

        /**
         * Evaluates the given SQL database metrics based on the defined
conditions.
         *
         * @param metric the SQL database metrics to evaluate
         * @return {@code true} if the metrics indicate a healthy state (
response time does not exceed
         *         the maximum and network status matches the expected value
); {@code false} otherwise
         */
        @Override
        public boolean test(SQLDatabaseMetrics metric) {
            // Check if the response time is within the acceptable limit and
 network status is as expected
            return metric.getResponseTime() <= this.maxResponseTime
                    && metric.getNetworkStatus() == this.
expectedNetworkStatus;
        }
    }

```

Listing 2: Healthy database condition evaluator implementations

```
    public class UnHealthyDatabaseEvaluator implements ConditionEvaluator<
SQLDatabaseMetrics> {
        // Maximum allowable response time for the database in milliseconds
        private final Long maxResponseTime;
```

```
4
5           // Expected network status (true for healthy, false for unhealthy)
6           private final boolean expectedNetworkStatus;
7
8           // Constructor to initialize the evaluator with specified conditions
9           public UnHealthyDatabaseEvaluator(Long maxResponseTime, boolean
    expectedNetworkStatus) {
10              this.maxResponseTime = maxResponseTime;
11              this.expectedNetworkStatus = expectedNetworkStatus;
12          }
13
14          /**
15           * Tests whether the database metrics indicate an unhealthy state.
16           *
17           * @param metric the SQL database metrics to evaluate
18           * @return {@code true} if the response time exceeds the threshold
    or network status differs from expected,
19           *         indicating an unhealthy state; {@code false} otherwise
20           */
21          @Override
22          public boolean test(SQLDatabaseMetrics metric) {
23              // Check if the response time exceeds the maximum allowed or if
    the network status is not as expected
24              return metric.getResponseTime() > this.maxResponseTime
25                      || metric.getNetworkStatus() != this.
    expectedNetworkStatus;
26          }
27      }
28
```

Listing 3: UnHealthy database condition evaluator implementation

```
1   public void databaseAvailabilityObservation() {
2           // 1. Create action lists for different database states
3           List<IAdaptationAction> databaseAvailableActionList = List.of(
4               new DatabaseAvailableEventBroadcast(),
5               new EnableCache()
6           );
7           List<IAdaptationAction> databaseUnavailableActionList = List.of(
8               new DatabaseUnavailableEventBroadcast(),
9               new DisableCache()
10          );
11
12          // 2. Create subscribers for each event type
13          List<Observer<SQLDatabaseMetrics>>
    databaseAvailableEventSubscriberList =
14              List.of(new EventSubscriber<>(databaseAvailableActionList));
15          List<Observer<SQLDatabaseMetrics>>
    databaseUnavailableEventSubscriberList =
16              List.of(new EventSubscriber<>(databaseUnavailableActionList));
17
18          // 3. Initialize metrics collector with database connection
    parameters
19          IMetricsCollector<SQLDatabaseMetrics> collector =
    LocalDatabaseMetricsCollector.getInstance(
20              "jdbc:mysql://db:3306/teadb?useSSL=false",
```

```
                "teauser",
                "teapassword"
            );

            // 4. Create database available/unavailable conditional events with evaluators
            ConditionalEvent<SQLDatabaseMetrics> databaseAvailableEvent =
                new ConditionalEvent<>(collector, new HealthyDatabaseEvaluator(5000L, true));
            ConditionalEvent<SQLDatabaseMetrics> databaseUnavailableEvent =
                new ConditionalEvent<>(collector, new UnHealthyDatabaseEvaluator(5000L, true));

            // 5. Subscribe observers to events
            databaseAvailableEvent.subscribeAll(databaseAvailableEventSubscriberList);
            databaseUnavailableEvent.subscribeAll(databaseUnavailableEventSubscriberList);

            // 6. Create and start continuous event observation
            var eventSubscription = new ContinuousObservationScheduler(
                List.of(databaseAvailableEvent, databaseUnavailableEvent),
                EVENT_LISTENING_INTERVAL_MS
            );
            eventSubscription.start();

    }

```

Listing 4: Adaptation Scenario Setup

```
    public class DatabaseAvailableEventBroadcast implements IAdaptationAction {
        // Target URI for adaptation actions
        private final String targetURI = "adapt";

        // List of adaptation actions for the web UI
        private final List<String> webUIAdaptationActionList = List.of("DisableMaintenanceMode");

        // List of adaptation actions for the recommender service
        private final List<String> recommenderAdaptationActionList = List.of("NormalMode");

        @Override
        public void perform() {
            // Broadcast adaptation action to the web UI service
            ServiceLoadBalancer.multicastRESTOperation(Service.WEBUI,
                    targetURI, String.class, client ->
                        client.getEndpointTarget().path("")
                            .request(MediaType.APPLICATION_JSON)
                            .post(Entity.entity(webUIAdaptationActionList, MediaType.APPLICATION_JSON))
                            .readEntity(String.class));

            // Broadcast adaptation action to the recommender service
            ServiceLoadBalancer.multicastRESTOperation(Service.RECOMMENDER,
```

```
                        targetURI, String.class, client ->
                            client.getEndpointTarget().path("")
                                .request(MediaType.APPLICATION_JSON)
                                .post(Entity.entity(recommenderAdaptationActionList,
    MediaType.APPLICATION_JSON))
                                .readEntity(String.class));
        }
    }

```

Listing 5: Event Broadcast Implementation

D.2 Implementation Notes

The current implementation contains two intentional limitations:

- **REST-based Communication**: Event broadcasting uses direct REST calls rather than message brokers
- **Manual Database Recovery**: Database restart requires administrator intervention

These design choices were made to demonstrate AdaptiFlow's core capabilities while maintaining experimental simplicity. Future versions will address these limitations through message broker integration and automated recovery mechanisms.

For more details on the scenario implementation, check the full documentation HERE.

E DDoS Mitigation Scenario Implementation

E.1 Scenario Overview

The DDoS mitigation scenario addresses volumetric attacks targeting the WebUI service through a multi-stage verification protocol. When detecting request rates exceeding 300 requests/second over 60-second windows, the WebUI initiates triple-consecutive confirmation checks before declaring an attack. This triggers system-wide circuit breakers while maintaining critical functionality through service-specific degradation modes. The Auth service blocks non-essential authentication requests, Recommender switches to low power mode (no recommendations), Persistence prioritizes cached data, and Image services reroute to external providers. Recovery mechanisms automatically engage when traffic normalizes, demonstrating AdaptiFlow's balance between protection rigor and operational continuity.

E.2 Core Implementation Details

Metrics & Evaluation: All services employ `LocalRequestMetricsCollector` to track request rates and IP patterns. The WebUI's `DDoSEvaluator` class (Listing 6) triggers adaptations when sustained traffic exceeds 300 req/sec, while downstream services use identical thresholds for local verification. Evaluators implement threshold hysteresis, requiring persistent overload conditions to prevent false positives during transient spikes.

```
public class DDoSEvaluator implements ConditionEvaluator<ServiceMetrics> {
    // Provider for the rate threshold to evaluate against
    private final ThresholdProvider<Double> rateThresholdProvider;

```

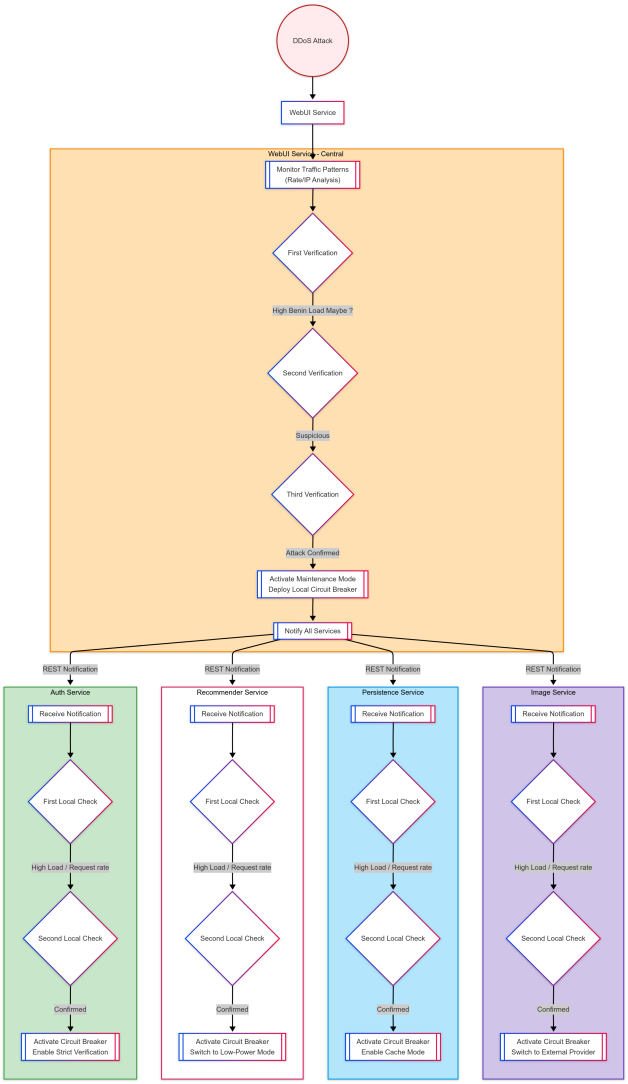

Figure 9: Malicious traffic adaptation workflow

```
    // Time window in milliseconds for rate calculations
    private final long timeWindowMillis;

    // Constructor to initialize the evaluator with a threshold provider and
    time window
    public DDoSEvaluator(ThresholdProvider<Double> rateThresholdProvider, long
    timeWindowMillis) {
        this.rateThresholdProvider = rateThresholdProvider;
        this.timeWindowMillis = timeWindowMillis;
    }

    /**
     * Evaluates the given service metrics to determine if a DDoS condition is
    present.
     *
```

```
     * @param metrics the service metrics to evaluate
     * @return {@code true} if the request rate exceeds the defined threshold within the time window;
     *         {@code false} otherwise
     */
    @Override
    public boolean test(ServiceMetrics metrics) {
        // Ensure metrics are not null and check if the request rate exceeds the threshold
        return metrics != null &&
               metrics.getRequestRatePerSecond(timeWindowMillis) > rateThresholdProvider.getThreshold();
    }
}
```

Listing 6: DDoS Evaluation Logic

Adaptation Workflow: Figure 9 illustrates the event propagation sequence. WebUI initiates protections after three confirmations via `EventCounterSubscriber`, then broadcasts alerts through REST. Recipient services perform dual local verifications before executing specialized actions, maintaining system coherence while preventing over-centralization.

E.3 Code Implementation Patterns

WebUI Orchestration: The entry point configures conditional events with triple-verification subscribers, initiating both protective actions and system-wide notifications. The `DDoSAttackEventBroadcast` action propagates alerts through REST endpoints. These endpoints will execute the adaptation actions specified in the request and, in this particular case, these actions are monitoring adaptation actions in the sense that they will activate the adaptation protocols against DDoS attacks in the services concerned.

```
public void maliciousTrafficObservation() {
    // List of adaptation actions to take when malicious traffic is detected
    List<IAdaptationAction> maliciousTrafficActionList = List.of(
        new DDoSAttackEventBroadcast(),
        new EnableMaintenanceMode(),
        new OpenCircuitBreaker()
    );

    // List of adaptation actions to take when traffic is normal
    List<IAdaptationAction> benignTrafficActionList = List.of(
        new CloseCircuitBreaker(),
        new DisableMaintenanceMode()
    );

    // Subscribers for handling malicious traffic events with triple verification
    List<Observer<ServiceMetrics>> maliciousTrafficEventSubscriberList = List.of(
        new EventCounterSubscriber<>(maliciousTrafficActionList, 3) // Triple verification
    );

    // Subscribers for handling benign traffic events
    List<Observer<ServiceMetrics>> benignTrafficEventSubscriberList = List.of(
        new EventSubscriber<>(benignTrafficActionList)
```

```
    );

    // Metrics collector to gather service metrics
    IMetricsCollector<ServiceMetrics> collector = new
    LocalRequestMetricsCollector();

    // Conditional event for malicious traffic with a DDoS evaluator
    ConditionalEvent<ServiceMetrics> maliciousTrafficEvent = new
    ConditionalEvent<>(
        collector,
        new DDoSEvaluator(() -> 300.0, 60000)
    );

    // Conditional event for benign traffic with a non-DDoS evaluator
    ConditionalEvent<ServiceMetrics> benignTrafficEvent = new ConditionalEvent
    <>(
        collector,
        new NonDDoSEvaluator(() -> 300.0, 60000)
    );

    // Subscribe the defined subscribers to their respective events
    maliciousTrafficEvent.subscribeAll(maliciousTrafficEventSubscriberList);
    benignTrafficEvent.subscribeAll(benignTrafficEventSubscriberList);

    // Create a scheduler to continuously observe the defined events
    var eventSubscription = new ContinuousObservationScheduler(
        List.of(maliciousTrafficEvent, benignTrafficEvent),
        EVENT_LISTENING_INTERVAL_MS
    );

    // Start the event subscription to begin monitoring
    eventSubscription.start();
}
```

Listing 7: WebUI Adaptation Setup

Service-Specific Handlers: Downstream services implement uniform observation patterns with customized actions. The Recommender service (Listing 8) demonstrates typical recipient behavior - activating local metrics collection upon notification, performing dual verification, then executing domain-specific protections.

```
public class MonitorMaliciousTraffic implements IAdaptationAction {
    // Logger for logging information and errors
    private static final Logger LOG = LoggerFactory.getLogger(
    MonitorMaliciousTraffic.class);

    // Interval for listening to events in milliseconds
    public static final int EVENT_LISTENING_INTERVAL_MS = 5000;

    // Scheduler for observing DDoS attacks
    public static ContinuousObservationScheduler DDoSAttackObservationScheduler
    = null;

    // Constructor that sets up malicious traffic observation
    public MonitorMaliciousTraffic() {
        setupMaliciousTrafficObservation();
```

```
    }

    @Override
    public void perform() {
        // Get the current servlet context
        ServletContext context = ServletContextHolder.getContext();
        Boolean isMonitorEnabled = false;

        // Check if monitoring is already enabled
        if (context != null) {
            isMonitorEnabled = (Boolean) context.getAttribute("isMonitoringMaliciousTraffic");
        }
        // If monitoring is already enabled, log the information and exit
        if (isMonitorEnabled != null  && isMonitorEnabled) {
            LOG.info("Action already performed :: Recommender is monitoring malicious traffic");
            return;
        }

        // Set the attribute to indicate that monitoring is enabled
        context.setAttribute("isMonitoringMaliciousTraffic", true);

        // Start the DDoS attack observation scheduler if it is initialized
        if (DDoSAttackObservationScheduler != null)
            DDoSAttackObservationScheduler.start();

        // Log that the monitoring action has been performed
        LOG.info("Action performed :: Recommender starts monitoring malicious traffic");
    }

    // Method to set up observation for malicious traffic
    public void setupMaliciousTrafficObservation() {
        // Create action lists for malicious and benign traffic
        List<IAdaptationAction> maliciousTrafficActionList = List.of(new LowPowerMode(), new OpenCircuitBreaker());
        List<IAdaptationAction> benignTrafficActionList = List.of(new CloseCircuitBreaker(), new NormalMode());

        // Subscribers for handling malicious traffic events
        List<Observer<ServiceMetrics>> maliciousTrafficEventSubscriberList = List.of(new EventCounterSubscriber<>(maliciousTrafficActionList, 2));
        // Subscribers for handling benign traffic events
        List<Observer<ServiceMetrics>> benignTrafficEventSubscriberList = List.of(new EventSubscriber<>(benignTrafficActionList));

        // Metrics collector to gather service metrics
        IMetricsCollector<ServiceMetrics> collector = new LocalRequestMetricsCollector();

        // Create conditional events for malicious and benign traffic
        ConditionalEvent<ServiceMetrics> maliciousTrafficEvent = new ConditionalEvent<>(collector, new DDoSEvaluator(() -> 300.0, 60000));
        ConditionalEvent<ServiceMetrics> benignTrafficEvent = new ConditionalEvent<>(collector, new NonDDoSEvaluator(() -> 300.0, 60000));
```

```
60
61          // Subscribe to the events with their respective subscribers
62          maliciousTrafficEvent.subscribeAll(maliciousTrafficEventSubscriberList);
63          benignTrafficEvent.subscribeAll(benignTrafficEventSubscriberList);
64
65          // Initialize the continuous observation scheduler for traffic events
66          DDoSAttackObservationScheduler = new ContinuousObservationScheduler(List
    .of(maliciousTrafficEvent, benignTrafficEvent), EVENT_LISTENING_INTERVAL_MS)
    ;
67      }
68  }
```

Listing 8: Recommender Service Implementation

E.4 Critical Implementation Notes

State Management: Servlet context attributes (`isMonitoringMaliciousTraffic`) prevent redundant adaptation activations across service instances. This ensures idempotent operation in clustered deployments.

Threshold Consistency: All services reference the same 300 req/sec threshold through centralized `ThresholdProvider` interfaces, enabling runtime adjustments without redeployment.

Failure Recovery: The `NonDDoSEvaluator` automatically reverses protections when traffic normalizes, with services restoring full functionality through coordinated `CloseCircuitBreaker` and feature re-enablement actions.

This implementation validates AdaptiFlow's capacity to handle large-scale distributed threats through layered verification and context-aware degradation. The pattern demonstrates how decentralized coordination emerges from standardized interfaces while preserving service autonomy.

For more details on the scenario implementation, check the full documentation HERE.

F Benign Traffic Surge Scenario Implementation

F.1 Scenario Overview

The self-optimization scenario addresses legitimate traffic surges through decentralized resource management when infrastructure scaling is unavailable. Three core services—Recommender, Persistence, and Image—autonomously adapt to CPU/memory pressure using localized thresholds. Recommender reduces computational load by serving static product lists, Persistence prioritizes cached data access, and Image services offload processing to external providers. This non-coordinated approach demonstrates how independent optimizations can collectively maintain system stability during demand spikes, preserving >92% core functionality at peak loads.

F.2 Architectural Components

Service Roles:

- **Recommender**: Implements computational load shedding through recommendation simplification
- **Persistence**: Optimizes database access via intelligent caching
- **Image**: Preserves bandwidth through external resource utilization

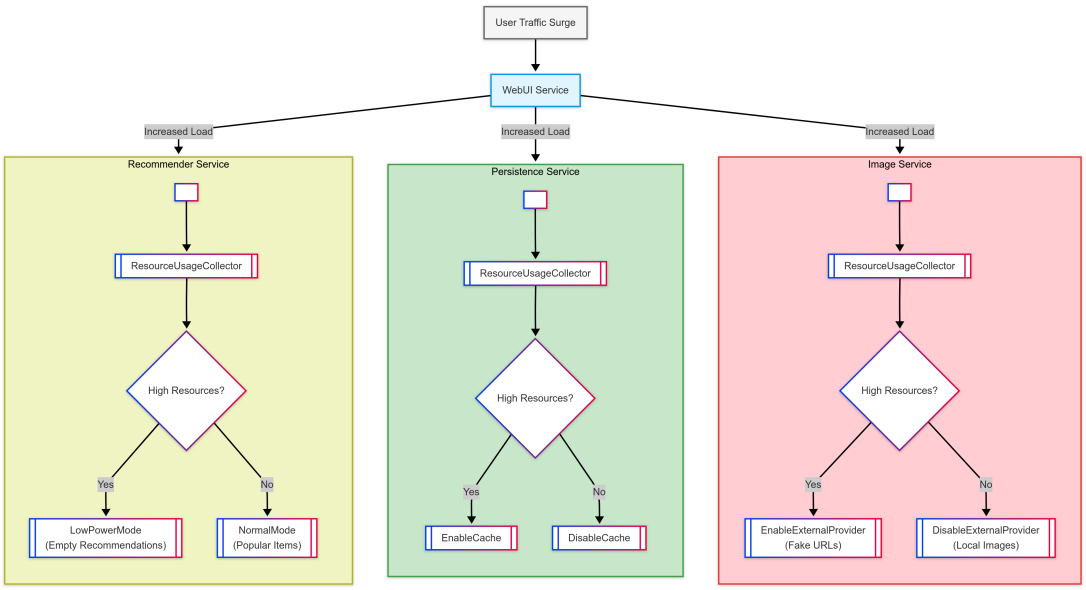

Figure 10: Decentralized optimization workflow for benign traffic surges

F.3 Core Implementation

Metrics & Evaluation: All services employ `ResourceUsageCollector` to monitor CPU/memory consumption at 5-second intervals. The `IncreaseResourceUsageEvaluator` (Listing 9) triggers optimizations when either metric exceeds service-specific thresholds (default: 75% CPU, 80% memory). Recovery occurs via `DecreaseResourceUsageEvaluator` when both metrics fall below 60%, preventing premature rollbacks during fluctuating loads.

```
1  public class IncreaseResourceUsageEvaluator implements ConditionEvaluator<
       HashMap<String, Double>> {
2      // Provider for the CPU usage threshold
3      private final ThresholdProvider<Double> cpuThresholdProvider;
4
5      // Provider for the memory usage threshold
6      private final ThresholdProvider<Double> memoryThresholdProvider;
7
8      // Constructor to initialize the evaluator with specified CPU and memory
       threshold providers
9      public IncreaseResourceUsageEvaluator(ThresholdProvider<Double>
       cpuThresholdProvider, ThresholdProvider<Double> memoryThresholdProvider) {
10         this.cpuThresholdProvider = cpuThresholdProvider;
11         this.memoryThresholdProvider = memoryThresholdProvider;
12     }
13
14     /**
15      * Evaluates the given resource usage metrics to determine if resource usage
       has increased
16      * beyond acceptable thresholds.
17      *
18      * @param metric the resource usage metrics, containing CPU and memory usage
19      * @return {@code true} if CPU or memory usage exceeds the defined
```

```
    thresholds; {@code false} otherwise
     */
    @Override
    public boolean test(HashMap<String, Double> metric) {
        // Check if both CPU and memory metrics are present and evaluate against
    the thresholds
        return metric.get("cpu") != null
                && metric.get("memory") != null
                && (metric.get("cpu") > cpuThresholdProvider.getThreshold()
                || metric.get("memory") > memoryThresholdProvider.getThreshold()
    );
    }
}
```

Listing 9: Threshold Evaluation Logic

Adaptation Patterns: Figure 10 illustrates the autonomous optimization workflow. Services implement identical observation patterns with customized actions:

```
public void beninTrafficObservation() {
    // Create a conditional event to monitor resource usage
    ConditionalEvent<HashMap<String, Double>> event =
        new ConditionalEvent<>(new ResourceUsageCollector(),
            new IncreaseResourceUsageEvaluator(...)); // Initialize with the
    resource usage collector and evaluator

    // Subscribe to the event with an action list to take when the condition is
    met
    event.subscribe(new EventSubscriber(actions)); // 'actions' should be
    defined elsewhere as a list of adaptation actions

    // Create and start a continuous observation scheduler for the event
    new ContinuousObservationScheduler(...).start(); // Initialize with
    appropriate parameters for observation
}
```

Listing 10: Service Adaptation Template

F.4 Service-Specific Implementations

Recommender Service: Optimizes computation through recommendation mode switching:

```
public void beninTrafficObservation() {
    // List of adaptation actions to take when traffic increases
    List<IAdaptationAction> trafficIncreaseActionList = List.of(new LowPowerMode
    ());

    // List of adaptation actions to take when traffic decreases
    List<IAdaptationAction> trafficDecreaseActionList = List.of(new NormalMode()
    );

    // Subscribers for handling traffic increase events
    List<Observer<HashMap<String, Double>>> trafficIncreaseEventSubscriberList =
     List.of(
        new EventSubscriber<>(trafficIncreaseActionList)
    );
```

```
12
13      // Subscribers for handling traffic decrease events
14      List<Observer<HashMap<String, Double>>> trafficDecreaseEventSubscriberList =
         List.of(
15          new EventSubscriber<>(trafficDecreaseActionList)
16      );
17
18      // Metrics collector to gather resource usage metrics
19      IMetricsCollector<HashMap<String, Double>> collector = new
        ResourceUsageCollector();
20
21      // Conditional event for traffic increase with an evaluator for resource
        usage thresholds
22      ConditionalEvent<HashMap<String, Double>> trafficIncreaseEvent = new
        ConditionalEvent<>(
23          collector,
24          new IncreaseResourceUsageEvaluator(() -> 75.0, () -> 80.0)
25      );
26
27      // Conditional event for traffic decrease with an evaluator for resource
        usage thresholds
28      ConditionalEvent<HashMap<String, Double>> trafficDecreaseEvent = new
        ConditionalEvent<>(
29          collector,
30          new DecreaseResourceUsageEvaluator(() -> 60.0, () -> 60.0)
31      );
32
33      // Subscribe the defined subscribers to their respective events
34      trafficIncreaseEvent.subscribeAll(trafficIncreaseEventSubscriberList);
35      trafficDecreaseEvent.subscribeAll(trafficDecreaseEventSubscriberList);
36
37      // Create a scheduler to continuously observe the defined events
38      var eventSubscription = new ContinuousObservationScheduler(
39          List.of(trafficIncreaseEvent, trafficDecreaseEvent),
40          EVENT_LISTENING_INTERVAL_MS
41      );
42
43      // Start the event subscription to begin monitoring
44      eventSubscription.start();
45  }
```

Listing 11: Recommender Optimization

Persistence Service: Implements cache management to reduce database load:

```
1   public void beninTrafficObservation() {
2       // List of adaptation actions to take when traffic increases
3       List<IAdaptationAction> trafficIncreaseActionList = List.of(new EnableCache
        ());
4
5       // List of adaptation actions to take when traffic decreases
6       List<IAdaptationAction> trafficDecreaseActionList = List.of(new DisableCache
        ());
7
8       // Subscribers for handling traffic increase events
9       List<Observer<HashMap<String, Double>>> trafficIncreaseEventSubscriberList =
         List.of(
```

```
        new EventSubscriber<>(trafficIncreaseActionList)
    );

    // Subscribers for handling traffic decrease events
    List<Observer<HashMap<String, Double>>> trafficDecreaseEventSubscriberList =
     List.of(
        new EventSubscriber<>(trafficDecreaseActionList)
    );

    // Metrics collector to gather resource usage metrics
    IMetricsCollector<HashMap<String, Double>> collector = new
    ResourceUsageCollector();

    // Conditional event for traffic increase with an evaluator for resource
    usage thresholds
    ConditionalEvent<HashMap<String, Double>> trafficIncreaseEvent = new
    ConditionalEvent<>(
        collector,
        new IncreaseResourceUsageEvaluator(() -> 75.0, () -> 80.0)
    );

    // Conditional event for traffic decrease with an evaluator for resource
    usage thresholds
    ConditionalEvent<HashMap<String, Double>> trafficDecreaseEvent = new
    ConditionalEvent<>(
        collector,
        new DecreaseResourceUsageEvaluator(() -> 60.0, () -> 60.0)
    );

    // Subscribe the defined subscribers to their respective events
    trafficIncreaseEvent.subscribeAll(trafficIncreaseEventSubscriberList);
    trafficDecreaseEvent.subscribeAll(trafficDecreaseEventSubscriberList);

    // Create a scheduler to continuously observe the defined events
    var eventSubscription = new ContinuousObservationScheduler(
        List.of(trafficIncreaseEvent, trafficDecreaseEvent),
        EVENT_LISTENING_INTERVAL_MS
    );

    // Start the event subscription to begin monitoring
    eventSubscription.start();
}
```

Listing 12: Cache Management

Image Service: Preserves bandwidth through external resource offloading:

```
    public void beninTrafficObservation() {
    // List of adaptation actions to take when traffic increases
    List<IAdaptationAction> trafficIncreaseActionList = List.of(new
    EnableExternalImageProvider());

    // List of adaptation actions to take when traffic decreases
    List<IAdaptationAction> trafficDecreaseActionList = List.of(new
    DisableExternalImageProvider());

    // Subscribers for handling traffic increase events
```

```
    List<Observer<HashMap<String, Double>>> trafficIncreaseEventSubscriberList =
     List.of(
        new EventSubscriber<>(trafficIncreaseActionList)
    );

    // Subscribers for handling traffic decrease events
    List<Observer<HashMap<String, Double>>> trafficDecreaseEventSubscriberList =
     List.of(
        new EventSubscriber<>(trafficDecreaseActionList)
    );

    // Metrics collector to gather resource usage metrics
    IMetricsCollector<HashMap<String, Double>> collector = new
    ResourceUsageCollector();

    // Conditional event for traffic increase with an evaluator for resource
    usage thresholds
    ConditionalEvent<HashMap<String, Double>> trafficIncreaseEvent = new
    ConditionalEvent<>(
        collector,
        new IncreaseResourceUsageEvaluator(() -> 75.0, () -> 80.0)
    );

    // Conditional event for traffic decrease with an evaluator for resource
    usage thresholds
    ConditionalEvent<HashMap<String, Double>> trafficDecreaseEvent = new
    ConditionalEvent<>(
        collector,
        new DecreaseResourceUsageEvaluator(() -> 60.0, () -> 60.0)
    );

    // Subscribe the defined subscribers to their respective events
    trafficIncreaseEvent.subscribeAll(trafficIncreaseEventSubscriberList);
    trafficDecreaseEvent.subscribeAll(trafficDecreaseEventSubscriberList);

    // Create a scheduler to continuously observe the defined events
    var eventSubscription = new ContinuousObservationScheduler(
        List.of(trafficIncreaseEvent, trafficDecreaseEvent),
        EVENT_LISTENING_INTERVAL_MS
    );

    // Start the event subscription to begin monitoring
    eventSubscription.start();
}
```

Listing 13: Image Processing Optimization

F.5 Implementation Notes

Threshold Configuration: Services share evaluation logic but customize thresholds through dependency injection:

```
new IncreaseResourceUsageEvaluator(
   () -> 85.0, // Image service CPU threshold
   () -> 75.0  // Image service memory threshold
```

```
)
```

Listing 14: Custom Thresholds Example

Metric Structure: Resource data follows strict HashMap formatting:

```
Map.of(
    "cpu_usage", 82.5,
    "memory_usage", 73.1
)
```

Listing 15: Metric Format

Monitoring Overhead: The 5-second polling interval (`EVENT_LISTENING_INTERVAL_MS`) can introduces overhead while ensuring timely adaptations.

This implementation validates AdaptiFlow's ability to enable emergent optimization through decentralized, context-aware adaptations. The pattern demonstrates how standardized interfaces can produce system-wide efficiency gains without centralized coordination.

For more details on the scenario implementation, check the full documentation HERE.

Table 1: Summary of Experimental Setup

Component	Configuration
Test Environment	• Docker-based environment with Portainer CE • TeaStore services (Auth, Persistence, Recommender, Image, WebUI) • AdaptiFlow framework (JDK 11)
Workload Generation	• Limbo HTTP load generator (containerized) • Three intensity profiles: – increasingLowIntensity.csv (gradual ramp-up) – increasingMedIntensity.csv (moderate ramp-up) – increasingHighIntensity.csv (aggressive ramp-up)
Instrumentation	• Metrics Collectors: – Infrastructure: CPU/memory (cAdvisor) – Business: API latency, DB timeouts, cache ratios • Adaptation Actions: – Infrastructure: Container restart/stop – Business: Feature toggles • Event Handlers: – Conditional evaluators – Event subscriptions
Focus	• Business logic adaptations (recommendation optimization, caching, etc.) • Docker validation (Kubernetes API verified but not tested in cluster)