Decoupling adaptive control in TeaStore

Eddy Truyen
DistriNet, KU Leuven, Leuven, Belgium
Eddy.Truyen@kuleuven.be

The Adaptable TeaStore specification and its experimentation platform extends the TeaStore microservice application by embedding a self-adaptive control loop within each service, enabling dynamic monitoring of runtime events and adaptation through modular effectors. While effective and in line with the desire to have a high decoupling between microservices, the proposed approach applies effectors for adaptation *per container* and *instantaneously*, whereas some effectors should be applied to all replicas of a *microservice deployment* in a *planned* approach. Behavioral variations activated by effectors may also require *invasive changes* in the core business functionality of the Teastore code. In this implementation discussion paper, we investigate how legacy modularization techniques, software architectural methods, and the operator pattern can decouple the control logic from the application. We discuss when their particular trade-off between fine-grained expressive adaptation and system-wide control and reuse of adaptation strategies can pay off.

1 Introduction

Self-adaptive software systems have been a long standing locus of research during the past decades [7, 1, 17]. The MAPE-K control loop [7], which stands for Monitoring-Analysis-Planning-Execution with Knowledge, has been the basis for investigation of self-adaptive systems in various computer science disciplines including software engineering [1, 17] and distributed systems [14].

Self-adaptation has also become increasingly critical with the widespread adoption of cloud-native computing. Autoscaling of microservices for performance efficiency is a typical example. However, self-adaptive behavior within the microservice itself has received less attention. *Adaptable TeaStore* [13] is an important initiative to revive the research on fine-grained adaptation of behavior within components. Its experimentation platform [11, 2] allows to configure MAPE-K control loops for specific microservices by specifying rules that monitor and filter for sequences of events, analyse whether it is necessary to adapt the internal behavior, and finally to execute these adaptations within the Teastore microservice by means of effectors.

1.1 Adaptable Teastore and its experimentation platform

The experimentation platform of the Adaptable Teastore case study relies on an Adaptation Library [2] that encapsulates a reusable implementation of the Monitor and Execution components of the MAPE-K control loop, thus enabling real-time adaptation to operational changes. It needs to be loaded inside each microservice as a modular component to make adaptation as fine-grained and as tailored as possible per microservice .

It is only needed to specify for each microservice (1) high-level rules that link specific filtered events to the execution of specific effectors and (2) effectors that make fine-grained behavioral changes to the running microservice using well-known design patterns such as Singleton, Strategy and Factory. The effectors are tightly coupled within each microservice in order to allow as expressive behavioral change

Submitted to:
WACA 2025

as possible. Typically, an effector changes the behavior of a microservice by changing a static variable within the microservice. This static variable defines which Strategy or Factory object must be selected when executing the code of the microservice. The desired configuration of the static variable can also be controlled using a REST API [13].

1.2 Motivation

A short investigation of the source code of experimentation platform [2], hereafter referred as Adaptable Teastore, gives the impression that adaptation rules cannot be easily applied to a *microservice deployment* that consists of multiple replicas. After all, there is no central third party that oversees multiple containers belonging to a microservice. The question is therefore if a better tradeoff could have been made between high expressive power for fine-grained adaptation on the one hand and system-wide control over multiple containers of the same microservice.

A second problem with Adaptable Teastore is a large degree of code duplication across the modules containing behavioral variations. Effectors set static variables that influence the selection of Class-based objects in the Strategy or Factory design pattern. However, these different alternative Class-based objects may mix code of the behavioral variation with the base code of the Teastore system. The question therefore arises how existing advanced modularization techniques can improve class inheritance to decrease code duplication by means of superimposition of behavioral variations.

1.3 Structure of the paper

This paper first reviews a number of methods for decoupling adaptation logic – Aspect-Oriented Programming, architecture-based adaptation and the operator pattern – and looks at Context-Oriented Programming for improving the modularization of behavioral software variations. Thereafter we outline three scenarios for improved decoupling: (1) reuse of adaptation rules, (2) modularization of behavioral variations, and (3) robust adaption of microservice deployment. Finally we present guidance for future implementation of new versions of Adaptable Teastore.

2 Background

The Adaptable TeaStore embeds a per-container control loop that continuously monitors internal events and triggers adaptation actions based on runtime observations. However, alternative approaches such as AOP, COP, architectural frameworks, and the operator pattern provide different means to externalize or modularize this functionality. This section introduces these methods and techniques with a focus on their known advantages and disadvantages.

2.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) provides a mechanism to modularize cross-cutting concerns by separating them from the core business logic [9]. In Aspect/J [8], the first and still most popular AOP extension for Java [12], *pointcuts* define join points in the application where specific actions (*advices*) are injected at runtime or compile time around these join points.

Advantages include: (1) separation of concerns: adaptation mechanisms such as monitoring, logging, or fault tolerance can be isolated from core functionalities, (2) precise event filtering: fine-grained control over where and when adaptations are triggered through declarative pointcut expressions, (3) non-invasive

changes: crosscutting behavior variations can be woven into classes without modifying the base source code of these classes.

Limitations are: (1) fragile pointcut problem: minor changes in the application's structure (e.g., method renaming) can silently break pointcut bindings, leading to adaptation failures, (2) debugging complexity: identifying the origin of dynamic behavior can be difficult, especially in large distributed systems.

2.2 Context-Oriented Programming

Context-Oriented Programming (COP) [6] promotes the idea that the behavior of software entities can vary dynamically based on the execution context. Context-dependent behaviors are organized into *layers*, which can be dynamically activated or deactivated.

Advantages include: (1) dynamic behavioral variation: allows seamless switching between different adaptation strategies depending on the context (e.g., high-load vs low-load behavior), (2) encapsulation of contextual variations: behavioral variations are modularized cleanly, improving code readability and maintainability, (3) fine-grained adaptation: specific functionalities or subsystems can be targeted without impacting unrelated parts of the system.

Limitations are: (1) scope management complexity: defining the scope and lifetime of context activation remains non-trivial, particularly in event-driven or multi-threaded systems, (2) performance overhead: frequent dynamic layer activation/deactivation can introduce runtime overhead, especially if not carefully optimized.

2.3 Architectural Self-Adaptation: Rainbow

The well-known Rainbow framework [1] separates adaptation concerns into an externalized control architecture following the MAPE-K model. The control architecture makes abstraction of the underlying system by means of an observation interface for monitoring the system and an effector interface for executing adaptations in the application.

Advantages include: (1) reuse of adaptation strategies: common monitoring and adaptation policies can be written once and applied across various systems, (2) application-agnostic monitoring: the system under adaptation only needs to expose observation points and effectors, (3) separation of roles: facilitates separate evolution of the adaptive control system and the target application.

Limitations are: (1) difficulty in fine-grained control: external controllers may lack the granularity necessary for low-latency, system-specific adaptations, (2) deployment and integration effort: requires a non-trivial setup to install probes, sensors, and effectors, and to model the system appropriately, (3) potential communication overhead: indirect adaptation actions through external channels may lead to latency compared to in-process adaptations.

2.4 Operator pattern

Operators [3] are specialized controllers that extend the REST API of a system with domain-specific knowledge. They observe resource changes and reconcile actual system state towards the desired state. This model, known as *level-based reconciliation*, differs fundamentally from traditional event-driven control loops.

In level-based reconciliation, the operator does not respond directly to individual events. Instead, it periodically (or reactively) examines the current system state and computes the necessary actions to drive

it toward the desired specification. This design ensures that the controller is idempotent and self-healing: missed events or transient failures do not result in missed adaptations, since the desired state remains authoritative.

This is in contrast to *event-based reconciliation*, where system behavior is modified in direct response to individual event notifications. While this model can be responsive and expressive, it often leads to brittle logic, increased state management complexity, and susceptibility to event loss.

In Kubernetes (K8s), operators provide a powerful framework for automating system behavior by reconciling the desired state declared in a Custom Resource (CR) with the actual state observed in the cluster.

Advantages are: (1) declarative adaptation: operators manage desired system states declaratively, aligning well with infrastructure-as-code practices, (2) isolated, independent control loops: each resource type (e.g., database, cache, microservice) has its own control logic, minimizing interference between adaptations, (3) native integration with Kubernetes ecosystem: benefit from built-in resilience (e.g., leader election, failover) and event-driven architecture of Kubernetes, (4) automatic scaling and healing: operators can naturally implement self-healing and scaling adaptations based on resource observations.

Limitations include: (1) complex operator development: building sophisticated operators in Kubernetes requires advanced expertise in controller development, (2) overhead for lightweight adaptations: for simple adaptations, the cost of writing and maintaining an operator may outweigh its benefits, (3) Limited fine-grained business logic: operators are primarily infrastructure-focused; injecting detailed business adaptation logic may lead to bloated designs.

Table 1 summarizes the main characteristics of each approach.

Technique	**Advantages**	**Limitations**
AOP	Separation of concerns, event filtering	Fragile pointcuts, debugging
COP	Dynamic adaptation, modular layers	Scope management, performance
Rainbow	Application-agnostic, reusable control	Fine-grained control, communication overhead
K8s operators	Isolated control, CRD-based APIs, scaling	Development complexity, business logic limits

Table 1: Comparison of modularization techniques for adaptive control

3 Possible decoupling scenarios

In this section we review some of our previous work on applying the above technologies and methods to decouple not only the adaptation logic but also to improve the modularization of behavioral variations.

3.1 Write adaptation software once, apply it to multiple applications

In our previous work [4], we presented an approach to constructing self-adaptive systems by combining the ideas behind Rainbow and AOP. We use aspects to modularize the MAPE-K adaptation logic. There are separate aspects for monitoring, analysis, planning, and execution of effectors. However, to make these aspects reusable for multiple applications, the ideas of Rainbow are applied: aspects only observe and modify an application model that makes abstraction of the application.

Each activity of the MAPE-K control loop is thus realized as a distinct aspect that intercepts specific join points in the system.

Figure 1 illustrates how each part of the MAPE-K loop is realized through aspects that are woven into different parts of the application, forming an adaptable fabric over the system. The application-agnostic model is bound to the application by means of the inter-type declaration construct of Aspect/J, which allows to extend classes with new interfaces. Pointcuts will convert application-specific events to generic Event objects to which the Monitoring aspects listen.

Monitoring aspects observe the Event objects of the application model and evaluate system states using the application model interfaces that have been added to application classes by means of inter-type declarations. When relevant events or states are detected, an IndicationEvent is created. Analysis aspects aggregate multiple of these IndicationEvent objects using the tracematch construct of the abc compiler of Aspect/J, which allows filtering over sequences of joint point occurrences. When a match is found with a sequence of events, an AdaptationEvent object is eventually constructed. Adaptation aspects listen to these AdaptationEvent objects and then modify the system's behavior accordingly. The resulting architecture allows the control logic to be distributed across the system in a loosely coupled, yet effective manner, as the use of AOP allows these concerns to be injected without requiring manual modification of the existing code base of the application.

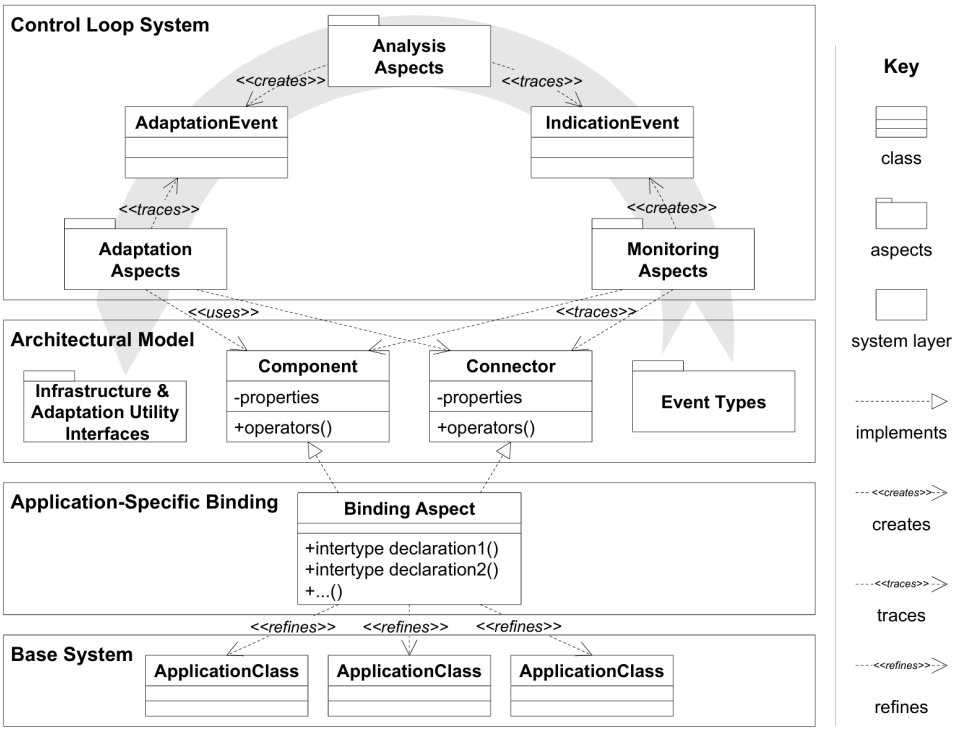

Figure 1: Aspect-oriented realization of the MAPE-K control loop (adapted from [4]).

The evaluation of this work shows that in comparison to an object-oriented version of a case study application, adaptation policies can be implemented more concisely, the resulting implementation code is better modularized and there is less coupling among the different implementation modules. As the pointcuts for event selection rely on a stable application model, this approach suffers less from the fragile pointcut problem. However, the modularization of adaptation policies into different aspects can complicate global reasoning and validation.

3.2 Encapsulate behavioral variations using COP

In another previous work [15], we present the application of Context-Oriented Programming (COP) to support runtime customization in multi-tenant Software-as-a-Service applications. In our model, behavioral adaptations specific to tenants or runtime contexts are encapsulated into *layers*, which can be dynamically activated or deactivated based on context such as user role, subscription tier, or environmental condition. A tenant can also activate multiple layers simultaneously

We implemented this approach using ContextJ [5], a Java-based COP language extension, and applied it to a multi-tenant hotel booking system deployed on Google App Engine (GAE). Context-dependent behaviors—such as interface customization, pricing models, or data access policies—are modularized into layers that can be consistently activated across application tiers, enabling uniform behavioral variation. These behavioral variations can be triggered by the current tenant ID as well as the tenant role who interacts with the application.

To illustrate the main constructs introduced by COP languages, we use this on-line hotel booking service. As we see in Figure 2, there are three tiers of the application: the presentation tier, the business tier, and the data tier. Now, we separate the behavior that is provided by the application into entities called *layer*. At the bottom, the *base* layer provides the core behavior of the application. In addition to this, we add the layers *LowSeason* and *VIP*. As can be seen, these layers modify the price calculation and/or the creation of bookings.

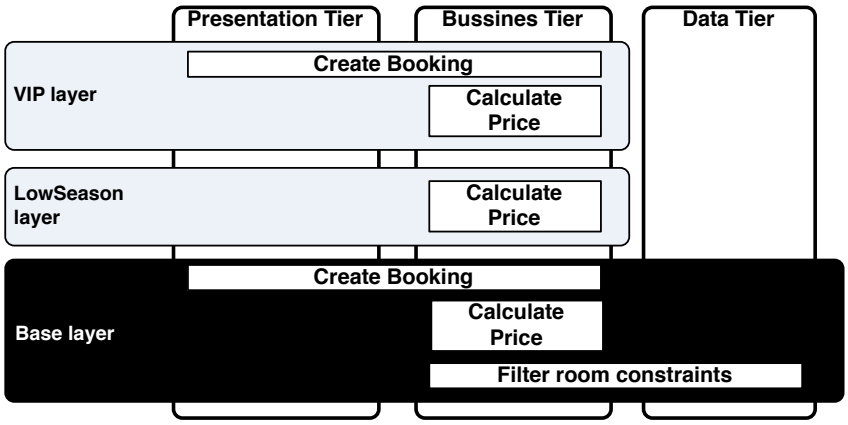

Figure 2: Behavior adaptations for the on-line booking application

The *layer* entity is a construct of COP. They are first-class entities that represent and group behavioral adaptations. In our example, we define the layers as separate classes, as shown in Listing 1.

```
1 public static final Layer VIP = new Layer();
2 public static final Layer LowSeason = new Layer();
```

Listing 1: Basic layer definition

After creating the classes, we define and associate the behavioral adaptations. In ContextJ, layer-specific behavior is defined by enclosing method declarations in `layer` expressions. Consider Listing 2, in which the `PriceCalculator` class is defined. This class provides a common method `calculateBookingPrice` that is part of the base layer. Then, we add specific method declarations for the `LowSeason` layer (Lines 8–12) and for the `VIP` layer (Lines 15–19).

```
public class PriceCalculator {
 ...
 public double calculateBookingPrice(double price,
  Date start, Date end) {
      //Calculate default price
  }

 layer LowSeason {
  public double calculateBookingPrice(double price,
      Date start, Date end) {
      //Calculate price with low season discount
   }
 }

 layer VIP {
  public double calculateBookingPrice(double price,
      Date start, Date end) {
      //Calculate price with VIP discount
     }
 }
}
```

Listing 2: Context-dependent behavior definition

Once behavior adaptations have been defined, they are made available to the application by activating the corresponding layer. In ContextJ, a layer can be activated explicitly by the `with` construct. This construct receives a block-like set of expressions. These expressions are compiled with the behavioral adaptations of the activated layer as their context. Outside of the `with` constructs, only the core behavior is active[1].

```
public class CreateBookingServlet
  extends HttpServlet {
  ...
 public void doPost(HttpServletRequest req,
    HttpServletResponse resp) throws IOException {
  ...
  with(getCurrentLayers(req)) {
     //set booking constraints
     createBooking(constrs, guest,
                   priceCalculator);
     //createBooking calls calculateBookingPrice()
    }

  public Booking createBooking(Constraints constrs,
     User guest,PriceCalculator priceCalculator) {
      //Create single booking
     //Display single booking
   }

  layer VIP {
```

[1]It is possible to nest `with` constructs.

```
21    public Booking createBooking(Constraints constrs,
22        User guest,PriceCalculator priceCalculator) {
23            //Create multiple bookings
24            //Display multiple bookings
25        }
26    }
27  }
```

Listing 3: Context-dependent behavior execution

Listing 3 shows this layer activation in ContextJ. Here, we see the definition of the `CreateBookingServlet`. In Line 7, we use `getCurrentLayers(req)` to determine the currently activated layers, for which we then execute the layer-specific `createBooking` method. For example, if the `VIP` layer is activated, its behavioral adaptation is applied; otherwise, the default behavior from the `Base layer` is retained. From Lines 20–26, we see the declaration of further layer-specific `createBooking` methods.

The configurations used for choosing a specific layer in the `getCurrentLayers(req)` method are stored as tenant metadata in the GAE datastore in the form of mappings from Tenant ID to a set of layers. Similarly, metadata about user-specific behavior adaptations can also be stored in this way in the GAE datastore. In case a specific tenant does not specify any preference, the behavior in the *Base layer* is selected.

COP offers several advantages in self-adaptive microservice systems. First, it supports *modular and reversible adaptations*, allowing adaptation logic to be defined orthogonally to core functionality. Second, layers can be activated *dynamically and per request*, aligning well with runtime context changes.

However, challenges arise when deploying COP in distributed systems. One key limitation is the lack of a built-in mechanism for ensuring *consistent layer activation between collaborating services*, which can lead to inconsistent states if multiple services interpret context independently. However, individual microservices are themselves decoupled from each other, only communicating with each other via their APIs. Therefore, independent context interpretation in different microservices is actually expected and desired.

Another problem is that the scope of layer activation is limited to a specific thread. This may cause problems in multi-threaded applications or event-based systems.

3.3 Robust adaptation using the operator pattern

We have applied the operator pattern to implement vendor-agnostic configuration of Kubernetes clusters in a cloud federation [16]. A desired cloud federation state is specified using a REST API and a reconciliation loop then ensures that the cloud federation is modified so that its actual state matches the desired state. A similar system-wide approach could also be implemented for microservice-agnostic adaption logic. We line out two possible scenarios.

3.3.1 Low power mode adaptation

In Adaptable Teastore, if one container of the Web UI microservices changes to low-power mode, this may require a consistent activation of low-power mode in all the other replicas of the Web UI microservice in order to ensure a consistent user experience. The first container can achieve this by invoking the REST adaptation endpoint of all relevant containers of the Web UI service.

Consider an operator that enables a low-power mode after detecting that one microservices is running out of memory. This can be realized using a Custom Resource (CR) with a boolean field in `spec` as shown below.

Listing 4: AdaptiveConfig Custom Resource Example

```
apiVersion: adaptive.io/v1alpha1
kind: AdaptiveConfig
metadata:
  name: example-config
spec:
  lowPowerAdaptation: false
status:
  outOfMemoryEvent: 2
```

The controller logic is written in the form of a reconciliation loop that counts the number of out of memory events and only sets the lowPowerAdaption to true after one or multiple of these events. Listing 5 shows how this can be implemented in Kubebuilder, a popular operator library for Kubernetes [10]. A clear disadvantage is the low-level code for maintaining the number of events (Line 16-22) and overhead with K8s API for storing the count (Line 9).

Listing 5: Controller reconciliation logic with event counting

```
1  func (r *AdaptiveConfigReconciler) Reconcile(ctx context.Context, req
       ctrl.Request) (ctrl.Result, error) {
2      var config adaptivev1.AdaptiveConfig
3      if err := r.Get(ctx, req.NamespacedName, &config); err != nil {
4          return ctrl.Result{}, client.IgnoreNotFound(err)
5      }
6
7      // Simulated external event detection
8      if detectOutOfMemory() {
9          config.Status.OutOfMemoryCount++
10         if err := r.Status().Update(ctx, &config); err != nil {
11             return ctrl.Result{}, err
12         }
13     }
14
15     // Trigger adaptation if threshold is reached
16     if config.Status.OutOfMemoryCount >= 3 && !config.Spec.
           LowPowerAdaptation {
17         config.Spec.LowPowerAdaptation = true
18         if err := r.Update(ctx, &config); err != nil {
19             return ctrl.Result{}, err
20         }
21         log.Log.Info("Enabled low power adaptation due to repeated out
               of memory events.")
22     }
23
24                 [... other code of Listing 6]
25
26 }
```

Listing 6: Controller reconciliation logic for activating low power mode

```
1                [...previous code of Listing 5]
2
3    // Reconcile actual microservice state with desired low power mode
4    if config.Spec.LowPowerAdaptation {
5        var podList corev1.PodList
6        if err := r.List(ctx, &podList, client.MatchingLabels{"app": "
             teastore"}); err != nil {
7            return ctrl.Result{}, err
8        }
9
10       for _, pod := range podList.Items {
11           // Query current power state (optional)
12                                                svc := pod.Status.PodIP
                                                     // or pod.Name if
                                                     using DNS
13           resp, err := http.Get(fmt.Sprintf("http://%s/adapt/
                 lowpowermode", svc))
14           if err != nil || resp.StatusCode != http.StatusOK {
15                                                    log.Log.Error(err, "
                                                         Failed to query
                                                         low power state",
                                                          "service", svc)
16               continue
17           }
18
19           var result struct {
20               LowPowerEnabled bool `json:"low_power_enabled"`
21           }
22           json.NewDecoder(resp.Body).Decode(&result)
23
24           if !result.LowPowerEnabled {
25               // Reconcile by enabling low power mode
26               http.Post(fmt.Sprintf("http://%s/adapt/lowpowermode/
                     enable", svc), "application/json", nil)
27               log.Log.Info("Reconciled low power mode for", "service",
                      svc)
28           }
29       }
30   }
31
32   return ctrl.Result{RequeueAfter: time.Minute}, nil
33 }
```

Setting lowPowerAdaption to True then triggers the same controller to bring the actual state of the relevant microservices into this desired state. It detects which microservices are still running in high-power mode and changes these to low-power mode (see Listing 6, Lines 10–29). Note that the reconciliation loop is robust for failed reconfiguration by rescheduling the controller again (cf. Line 32). Thus, in case some microservices are still running in high-power mode after a first reconfiguration attempt, this will be automatically detected by the controller, which then retries setting them to low-power mode. Note that a modified version of the `adapt` API endpoint of the Adaptable Teastore may be needed in order to

support a GET of the current state of the adaptation.

3.3.2 Planning adaptations

Another example where the operator pattern may come useful is when switching the Recommender algorithm. The adaptation logic immediately sets a new Recommender algorithm, requiring resetting the state of the old Recommender algorithm. But this adaptation logic does not take into account the progress of execution of the old Recommender algorithm: if the old version is serving a high number of requests, it is worth to defer switching to the new version until the request rate has dropped. In an operator, setting the Recommender algorithm is a modification of the desired state and it is possible to program the controller so that it brings the actual system into that desired state when specific systems state conditions are satisfied.

This example demonstrates how adaptation policies can be implemented in a robust, declarative manner. The CR extensions of the K8s API acts as a persistent, inspectable representation of adaptation state, while the controller enforces convergence toward the declared target configuration. Note, even without Kubernetes, this approach can be applied. For example, the registry API of Teastore could offer an API for managing the states of system-wide adaptations that must be applied to all pods. Alternatively, each Teastore microservice can have a leader replica that runs the controller in active mode whereas the adaptation state is stored in a persistent volume that is accessible by all replicas.

4 Discussion

This paper has reviewed and illustrated a number of approaches for decoupling the adaptive control logic from the Adaptable Teastore code. The motivation for this was three-fold:

(1) Write architectural adaptation rules once, but apply them to multiple applications automatically without requiring manual code changes.

(2) When behavioral variations cannot be cleanly encapsulated using object-oriented programming, advanced modularization techniques can be used to improve maintenance of the original TeaStore code.

(3) The scope of a single container may be too limited for some adaptations. Even if adaptation logic is written for a specific microservice, effectors may still need to be applied to all replicas of that microservice. For example, a low-power mode should be applied to all replicas of the WebUI microservice to ensure consistent user experience.

Techniques and methods such as AOP, COP, Rainbow, and the operator pattern offer viable alternatives to externalize and modularize adaptive control. Each presents distinct trade-offs between specificity, maintenance, and complexity. In summary we believe that Adaptable Teastore may adopt some ideas from the operator pattern without loosing the expressive power of its AdaptableLibrary. A central TeaStore microservice such as the Registry may take up the role implementing a central REST API for AdaptiveConfig as in Listing 4 and a central reconciliation loop as in Listing 6.

COP can be useful for modularizing behavioral variations of class objects provided that per thread activation of layers will work for that class. At every point in the application code where static variables, encapsulating the adaptation state, are consulted, it should be investigated on a case-by-case basis whether layer activation per thread will work.

AOP can be useful to separate the entire adaptation package from the original code of the different Teastore microservies. This enables smooth reuse of the adaptation package across multiple microservices and also caters for independent evolution of the Teastore code base. However, it is important

that aspects are based on stable pointcut abstractions and type interfaces that can be easily wrapped to Teastore classes.

References

[1] Shang-Wen Cheng, An-Cheng Huang, D. Garlan, B. Schmerl & P. Steenkiste (2004): *Rainbow: architecture-based self-adaptation with reusable infrastructure*. In: *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 276–277, doi:10.1109/ICAC.2004.1301377.

[2] Adaptable TeaStore Contributors (2025): *Adaptable TeaStore Experimentation Platform*. `https://gitlab.inria.fr/adaptable-teastore/experimentation-platform/`. Accessed: 2025-05-02.

[3] CoreOS & The Kubernetes Authors (2016): *Kubernetes Operators: Automating the Container Orchestration Platform*. `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/`. Accessed: 2025-04-29.

[4] Robrecht Haesevoets, Eddy Truyen, Tom Holvoet & Wouter Joosen (2010): *Weaving the Fabric of the Control Loop through Aspects*. In Danny Weyns, Sam Malek, Rogério de Lemos & Jesper Andersson, editors: *Self-Organizing Architectures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 38–65.

[5] Robert Hirschfeld (n.d.): *Context-Oriented Programming*. `https://www.hpi.uni-potsdam.de/hirschfeld/projects/cop/`. Accessed: 2025-05-05.

[6] Robert Hirschfeld, Pascal Costanza & Oscar Nierstrasz (2008): *Context-Oriented Programming*. *Journal of Object Technology* 7(3), pp. 125–151, doi:10.5381/jot.2008.7.3.a4. Available at `https://www.jot.fm/contents/issue_2008_03/article4.html`.

[7] J.O. Kephart & D.M. Chess (2003): *The vision of autonomic computing*. *Computer* 36(1), pp. 41–50, doi:10.1109/MC.2003.1160055.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William G. Griswold (2001): *An Overview of AspectJ*. In Jørgen Lindskov Knudsen, editor: *ECOOP 2001 — Object-Oriented Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 327–354.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier & John Irwin (1997): *Aspect-Oriented Programming*. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, *Lecture Notes in Computer Science* 1241, Springer, pp. 220–242, doi:10.1007/BFb0053381.

[10] The Kubernetes SIG API Machinery (2018): *Kubebuilder: SDK for building Kubernetes APIs using CRDs*. `https://book.kubebuilder.io`. Accessed: 2025-04-29.

[11] Zemtsop Ndadji, Brice Arléon, Bliudze Simon & Quinton Clément (2025): *Case Study: Adaptable Teastore (Spec. and Experimentation Platform)*. `https://waca-ws.github.io/2025/cfp/`.

[12] Tuan Nguyen (2024): *Intro to AspectJ*. Available at `https://www.baeldung.com/aspectj`. Accessed: 2025-05-02.

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

[14] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho & David Leslie (2016): *REX: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems*. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, USENIX Association, Savannah, GA, USA, pp. 333–348. Available at `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter`.

[15] Eddy Truyen, Nicolás Cardozo, Stefan Walraven, Jorge Vallejos, Engineer Bainomugisha, Sebastian Günther, Theo D'Hondt & Wouter Joosen (2012): *Context-oriented programming for customizable SaaS applications*. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, Association for Computing Machinery, p. 418–425, doi:10.1145/2245276.2245358.

[16] Eddy Truyen, Hongjie Xie & Wouter Joosen (2023): *Vendor-Agnostic Reconfiguration of Kubernetes Clusters in Cloud Federations*. Future Internet 15(2), doi:10.3390/fi15020063. Available at https://www.mdpi.com/1999-5903/15/2/63.

[17] Danny Weyns (2020): *An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*. Wiley. Available at https://onlinelibrary.wiley.com/doi/book/10.1002/9781119574910.