Adaptable TeaStore: A Choreographic Approach*

Giuseppe De Palma Saverio Giallorenzo Ivan Lanese Gianluigi Zavattaro
Università di Bologna and INRIA
Bologna, Italy
{giuseppe.depalma2,saverio.giallorenzo2,ivan.lanese,gianluigi.zavattaro}@unibo.it

Adaptable TeaStore has been recently proposed as a reference model for adaptable microservice architecture. It includes different configurations, as well as scenarios requiring to transition between them. We describe an implementation of the Adaptable TeaStore based on AIOCJ, a choreographic language that allows one to program multiparty systems which can adapt at runtime to different conditions. Following the choreographic tradition, AIOCJ ensures by construction correctness of communications (e.g., no deadlocks) before, during, and after adaptation. Adaptation is dynamic, and the adaptation scenarios need to be fully specified only at runtime. Using AIOCJ to model the Adaptable TeaStore, we showcase the strengths of the approach and its current limitations, providing suggestions for future directions for refining the paradigm (and the AIOCJ language in particular), to better align it with real-world Cloud architectures.

1 Introduction

The Adaptable TeaStore has been recently proposed [1] as a reference model for adaptable microservices architectures. It extends the TeaStore reference model [7] for static microservices architectures with multiple configurations as well as many adaptation scenarios that trigger transitions between them.

In this paper, we model the Adaptable TeaStore specification as adaptable choreographies, using AIOCJ (Adaptable Interaction-Oriented Choreographies in Jolie) [3, 2, 4], an executable language that allows one to program adaptable multiparty systems.

Following the tradition of choreographic specifications and programming [6, 8], in AIOCJ, a single artefact defines a multiparty system and one can derive the code of each component that participate in that system from the artefact via a projection operation. More precisely, the idea behind AIOCJ is to program an adaptable architecture where the participants each connect to multiple services and, when adapting the system, the participants would consistently execute new code (as in, "not present in the original choreography") that implement the adapted behaviour, possibly reconfiguring the connections among the participants and the services they have access to and changing the ways they use them.

The choreographic approach ensures by construction relevant correctness properties of communication, such as deadlock freedom. In the specific case of AIOCJ, these properties hold before, during, and after adaptation. As mentioned, a distinctive trait of AIOCJ programs is that they can adapt to behaviour unexpected at the time of writing the choreography. Concretely, when one programs an adaptable choreography, they need to specify which parts of the code may change in the future. The new code replacing the adaptable parts, and the definition of the conditions triggering the adaptation, can be added afterwards — in particular, while the original system is running. Adaptation works by specifying adaptation rules,

*Work partially supported by French ANR project SmartCloud ANR-23-CE25-0012, by PRIN project FREEDA (CUP: I53D23003550006) funded by the frameworks PRIN (MUR, Italy) and Next Generation EU, by project PNRR CN HPC - SPOKE 9 - Innovation Grant LEONARDO - TASI - RTMER funded by the NextGenerationEU European initiative through the MUR, and by INdAM - GNCS 2024 project MARVEL, code CUP E53C23001670001

Submitted to:
WACA 2025

which can be applied depending on the state of the system and of its execution context/environment. A single adaptation rule may change the code of multiple participants in a coordinated way.

Since we show an application of AIOCJ on the Adaptable TeaStore and our focus is on the modelling of the architecture of the case study (and not, e.g., its computational part), we use minimal implementations of the needed services, seen (in AIOCJ) as external services. In doing so, we follow a programming style [5] where AIOCJ provides adaptable connectors which coordinate the external services, as well as adaptation rules allowing such coordination to change depending on different needs[1].

2 Adaptable TeaStore: An Overview

While we refer the reader to the official Adaptable TeaStore specification [1] for full details, in this section, we briefly summarise the Adaptable TeaStore architecture to illustrate the components at play and the deployment modalities we choose for our AIOCJ implementation. We report the schema of Adaptable TeaStore's architecture from the specification [1] in Figure 1.

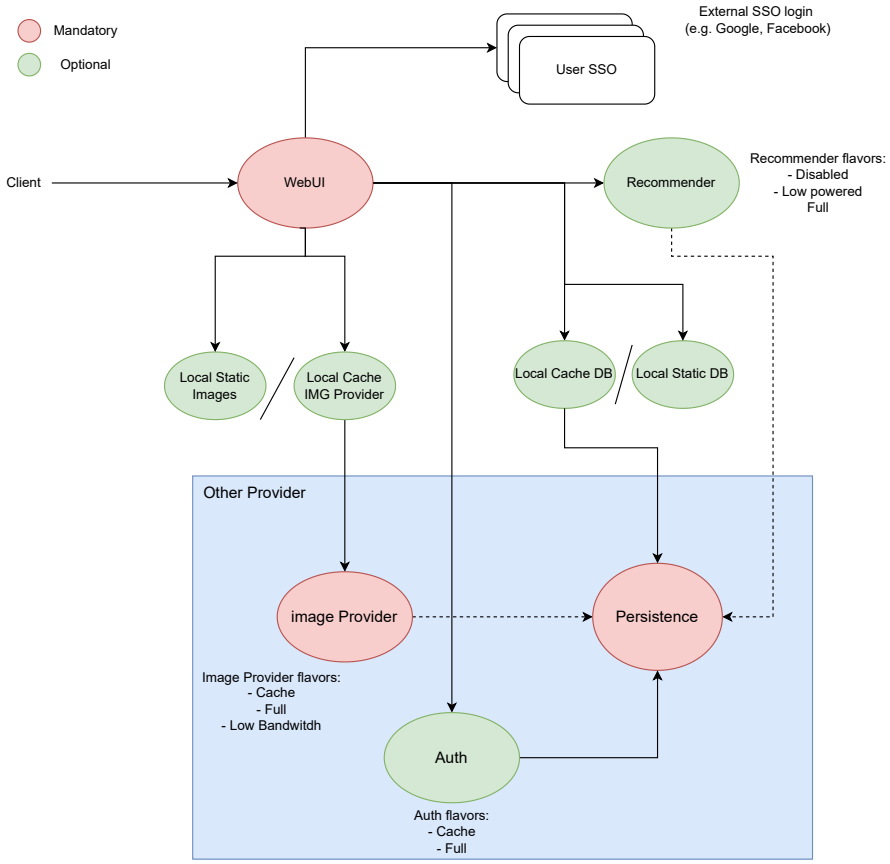

Figure 1: Adaptable TeaStore Architecture Diagram.

[1]While, in principle, one could implement a whole architecture in AIOCJ, to separate concerns, we follow a style that implements the coordination among components in AIOCJ and uses external services for the components' business logic.

The Adaptable TeaStore's architecture comprises 5 main services: WebUI, Auth, Persistence, Image Provider, and Recommender. The WebUI service works as the entry point for users, and it orchestrates all other services. The Persistence service acts as a layer on top of the database. We deploy it on a provider different from the local one, making it accessible via a Local Cache. When unavailable, a Local Static Database (with limited functionalities) replaces it. The WebUI uses the Persistence Service to retrieve and store data, while the Auth service uses it to retrieve user data, which it then passes to the WebUI.

Both the Image Provider and Recommender connect to the Persistence service. However, they only require this connection on startup (dashed lines) — since this startup syncronisation does not introduce new interaction scenarios than the main one, we omit to model this part in our implementation. The Image Provider must generate an image for each product and, like Persistence, operates remotely and connects via a Local Cache. Local Static Images can replace the Image Provider when needed. Auth provides authentication facilities and can rely on external User SSO services to enable authentication via, e.g., Google or Facebook — similarly to the startup synchronisation routine, we omit to model SSO interactions in our implementation to focus on the main behaviour of the application.

The Recommender offers users suggestions about potentially interesting products. The full-power version of the Recommender uses machine learning techniques associated with specific user preferences, but a low-powered version based on generic item popularity can replace it. The Recommender service is not an essential service, and can be completely disabled. Similarly, the Image Provider and the Auth service also offer different variants.

3 "Barebone" TeaStore, in AIOCJ

We start by presenting an AIOCJ implementation of the most basic version of (Adaptable) TeaStore, which we call "Barebone" TeaStore. The Barebone version only includes the essential services and interactions that offer minimum functionality to users, i.e., those provided by the WebUI, the Local Static Images, Local Static Persistence services. In this section, we start by introducing a non-adaptable version of Barebone TeaStore and, in the subsequent sections, we proceed to present and refine the implementation with the necessary AIOCJ constructs to make our program adaptable and able to cover most of the Adaptable TeaStore architectural specification.

We report in Listing 1 the AIOCJ code of Barebone TeaStore, focussing on the interesting part at lines 7–25, contained within the `aioc` scope, delimited by curly brackets. Therein, we find a choreography specifying the global behaviour of four participants: the user, marked `U`, the WebUI `W`, the Persistence service `P`, and the Images service `I`. The choreography represents user interactions using the `getInput` functionality, which presents the user with a prompt through which they can insert data. Specifically, the first action in the choreography concerns the `User` providing the address of the page of the TeaStore they want to visit; the data inserted by the `User` is stored in its local variable `address`. Since an AIOCJ program describes data located at different participants, it uses the `@`-notation to indicate which participant owns a given variable — in the case of the address, it is `address@U`. Then, we find the first interaction between participants of the choreography, where the `User` sends the address to the `WebUI`. Following standard practice of process calculi, all AIOCJ interactions are labelled — in this case, the label is `getPage`. The notation `U( address ) -> W( address )` means that `U` sends the value contained in its local variable `address` to `W`, which stores it in its local variable with the same name. Note the `;` between the first and second instructions. That particle is a sequential composition operator that specifies that the instruction on its left must execute before the instruction on its right. Since the system described by an AIOCJ program is distributed, to ensure the exact execution of sequential compositions

the AIOCJ compiler makes sure that there is at least one participant in common between two instructions composed in sequence — e.g., U is the common participant between the two first instructions — to guarantee the faithful implementation of the causal relation among the global actions in the choreography, which become distributed once compiled into different programs. Then, we open a scope at lines 10 and 22, so that we implement a fork-join pattern for parallelising the interaction between the WebUI and resp. the Persistence and the Images services. We implement the forking part of the patter with two internal scopes (resp. at lines 11–15 and 17–21) that we compose using the parallel composition operator `|`. These two scopes execute in parallel and, once terminated, join the larger sequential composition at the closure of the wrapping scope at line 22. In the internal scopes, the WebUI separately interacts with the Persistence and the Images services to obtain the textual content of the page, saved in the variable `info` of the WebUI, and the images of the page, saved in the `img` variable of the WebUI. In each scope, we resp. find the Persistence and Images participants that interact with the available APIs of the TeaStore services (or wrappers thereof), resp. `getPageInfo` and `getPageImage`, to obtain the contents of the page. In AIOCJ, developers can introduce the availability of APIs through the `include` instruction, found at the beginning of AIOCJ programs, e.g., in Listing 1, at lines 1–3 — where one can specify the address and communication medium of the service (e.g., `"socket://..."`) and the data format (e.g., `"soap"`).

After the join, the WebUI continues by calling the `compilePage` API to obtain the page, which it sends to the User.

Closing the example, we notice the presence of the `preamble` clause at line 5, which developers can use to specify configuration parameters, like the addresses of the participants. In this minimal example, we let that AIOCJ compiler assign the addresses, and we just indicate which, among the choreography's participants, is the `starter` — this information is necessary to implement a rendezvous procedure, where a `starter` participant, in this case, the WebUI, waits for all the other participants to contact it and then notifies them they can start the choreography.

```
1  include getPageInfo from "socket://localhost:8001" with "soap"
2  include getPageImg from "socket://localhost:8002" with "soap"
3  include compilePage from "socket://localhost:8000" with "soap"
4
5  preamble { starter: W }
6
7  aioc {
8   address@U = getInput( "Insert address" );
9   getPage: U( address ) -> W( address );
10  {
11   {
12     getPageInfo: W( address ) -> P( address );
13     info@P = getPageInfo( pid );
14     getInfo: P( info ) -> W( info )
15   }
16   |
17   {
18     getPageImg: W( address ) -> I( address );
19     img@I = getPageImg( address );
20     getImg: I( img ) -> W( img )
21   }
22  };
23  page@W = compilePage( info, img );
24  getPage: W( page ) -> U( page )
```

```
25 }
```

Listing 1: AIOCJ Choreography of the Barebone TeaStore version.

Structuring Adaptation We close this section by showing how one can introduce structured, distributed adaptation into AIOCJ programs. Specifically, we consider the case where other APIs for the compilation of the page could become available in the future.

Thanks to scopes, AIOCJ allows developers to adapt the behaviour of the participants in a choreography at runtime by integrating code written even after they started their execution.

To illustrate this feature, we modify the choreography from Listing 1 by replacing line 23 with the code in Listing 2. In general, an adaptation scope has two elements: a controller, which is indicated with the @-notation next to the scope, in Listing 2, the controller is the WebUI, and the body of the scope, contained within curly brackets, which delimits a piece of the choreography can change at runtime through the application of AIOCJ rules — discussed in the next sections.

At runtime, the controller is the participant in charge of selecting which rule applies to the scope. While this example is simple, i.e., the only participant in the scope is also the controller, a scope can regard multiple participants, which the controller coordinates to make sure they all follow the same piece of choreography found in a selected adaptation rule. When a coordinator reaches the beginning of a scope, it queries AIOCJ's runtime for adaptation rules to apply. The runtime queries the available rule repositories sequentially which, in turn, check the applicability condition of each of their rules. The runtime applies the first rule whose applicability condition holds, if any, by sending to the coordinator the code for each participant, which it distributes to the involved roles. In each role, the new code replaces the original one. If no rule applies, the coordinator tells to the other roles to execute the original code. This protocol ensures the consistency of adaptations, which derive from the fact that there is a single source of "truth" that determines the unfolding of the adaptation (the coordinator). Indeed, since rules can change at runtime (they can appear and disappear), having only the coordinator observe which rules are available solves the inherent problem of inconsistencies (roles could see different sets of rules at different times and locations) about rule availability and distribution of the related adaptation code.

Developers can associate properties with scopes to make the application of adaptation rules more precise, e.g., in Listing 2, we add the property tag, prefixed by N[2][3], that labels the scope as carrying the code for the "page_compiler". Scope properties are meant to describe the current implementation of the scope, including both functional and non-functional properties. Such properties are declared by the programmer, and the system only uses them to evaluate the applicability condition of adaptation rules, to decide whether a given adaptation rule can be applied to a given scope.

```
1  scope @W {
2   page@W = compilePage( info, img )
```

[2]Originally, the prefix stood for "non-functional", meaning that scope-level properties should capture non-functional properties. However, in practice, adaptation decisions often depend on interwoven functional and non-functional concerns.

[3]The prefixing of scope-level properties/variables is not necessary, since these are syntactically and contextually distinct from other variables (e.g., of the scope controller). On the contrary, the prefix is fundamental in rules, which can also consider other kinds of properties in applicability conditions. For symmetry, AIOCJ imposes the usage of the prefix also in scopes.

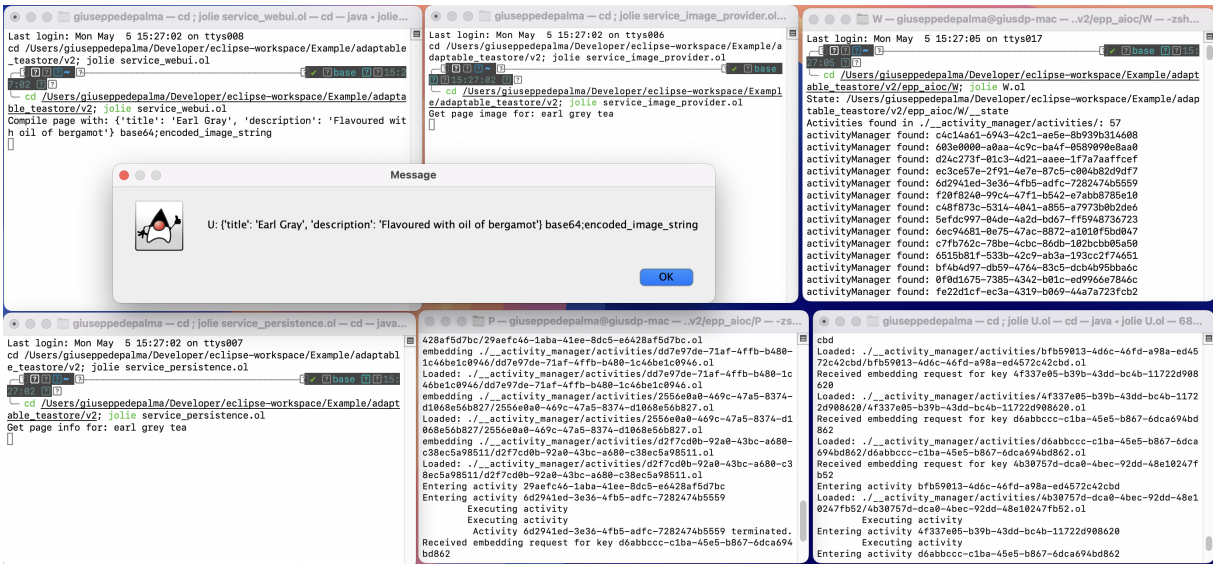

Figure 2: Execution of the Barebone TeaStore choreography.

```
3  } prop { N.tag = "page_compiler" };
```

Listing 2: Excerpt of AIOCJ Choreography of an adaptable version of Barebone TeaStore.

We further concretise our description by reporting in Figure 2 a screenshot of the (local) execution of the Barebone TeaStore choreography.

In the figure, we find a multi-terminal setup that demonstrates the execution of services built with AIOCJ. Specifically, we show the behaviour of Barebone TeaStore after the user inputs an address.

At the top centre of the image, a pop-up window appears showing the result of the user interaction: a tea product with the title “Earl Gray”, its description, and an associated base64 encoded image. This output results from the service invocations, e.g., to the Persistence and Image Provider, found within the choreography.

Each terminal window runs a different Jolie [9] service (launched via the “jolie” command, which executes the interpreter of Jolie programs). Besides the services that emulate/wrap the behaviour of the existing Adaptable TeaStore services (e.g., the Persistence service), the other ones are automatically generated by AIOCJ’s compiler, which produces both the services that implement the logic of the choreography participants and the services that constitute the (distributed) runtime environment for AIOCJ’s applications (e.g., an Environment service that lets administrators specify parameters of the application execution context).

In the figure, after inputting the address, the participants interact both among themselves and with the external services (through their APIs) to retrieve the content of the page requested, compile it, and show the result to the user.

Using Scopes to Implement Adaptation Summarising, AIOCJ provides scopes as a way to specify which parts of the application can be adapted. As a consequence, deciding which parts of the code to enclose in scopes is a relevant and non-trivial decision. Intuitively, one should enclose into scopes parts

of the code that may need to be adapted in the future. These parts include, e.g., the code modelling business rules that may need to change according to changed business needs, but also parts which are location dependent and parts which are relevant for performance or security reasons.

Of course, one could cut this Gordian Knot by having either a scope that encloses the whole choreography or many scopes covering all instructions. However, both solutions have relevant drawbacks. An all-encompassing scope would replace the whole choreography, which means that one has to treat adaptation as a monolithic change that has to integrate all relevant adaptation aspects (e.g., coalescing orthogonal functional and non-functional aspects, like security and availability), giving little-to-no support for modularity. Using per-instruction scopes is hardly a solution too, since adaptation may cover more than one scope and, at the moment, AIOCJ does not provide ways to structure multi-scope adaptation behaviours — as discussed in the next sections, Adaptable TeaStore requires one such kind of coordination, which we implicitly implement via bookkeeping.

4 Barebone TeaStore Choreography with Adaptable Recommender

As mentioned, Adaptable TeaStore allows for an optional Recommender, which comes in two flavours: low-power, based on item popularity, and full-power, using machine learning and user preferences.

To support such an adaptation in AIOCJ (like any other adaptation) we need two ingredients, a `scope` specifying where in the code adaptation should happen, and one or more adaptation `rule`s, specifying which new code should be used in case adaptation happens. Notably, adaptation rules can change (be provided, removed) at runtime, during the execution of the original choreography.

In this specific case, we use two `scope`s, the first one, introduced in the previous section, about page compilation, which allows one to exploit information from the Recommender to produce a page for the user, including recommendations, and one, shown in Listing 3, line 14. Notably, we put this new `scope` in parallel (`|`) with the computation of the page information and the page image, as discussed in the previous section.

An alternative would be to use just the `scope` for page compilation described in the previous section, with a rule introducing the Recommender as well. While conceptually simpler, this alternative would be less efficient since recommendation computation would be started after the end of the ones on page and image information.

```
1  {
2    {
3      getPageInfo: W( address ) -> P( address );
4      info@P = getPageInfo( pid );
5      getInfo: P( info ) -> W( info )
6    }
7    |
8    {
9      getPageImg: W( address ) -> I( address );
10     img@I = getPageImg( address );
11     getImg: I( img ) -> W( img )
12   }
13   |
14    scope @W { skip } prop { N.tag = "recommender" }
```

```
15 };
```

Listing 3: Adaptable TeaStore barebone version, Recommender only

We now describe the two adaptation rules, reported respectively in Listing 4 and Listing 5, starting from the former.

Adaptation rules have a main building block identified by the `do` clause, which specifies the code that needs to be executed in case adaptation is performed, replacing the code inside the scope under adaptation. However, rules include other elements as well. First, they may include additional external services (e.g., `getTopItems`) and additional participants, the latter introduced by keyword `newRoles`. Finally, the keyword `on` introduces the applicability condition of the rule: when a scope is met during execution, all available rules are checked for applicability (in no specific order). The first one whose condition evaluates to true is applied. Conditions may refer to properties described by the `scope`, prefixed by `N`, as in `N.tag == "recommender"`, and to properties described by the environment, prefixed by `E`, as in `E.recommender == "low-power"`. The former are meant to ensure that a rule is applied to the "correct" scope (i.e., the one refined by the addition of the related property), while the latter allow rules applicability to depend on environmental condition, e.g., if we are in an environment with limited power availability, so that the rule in Listing 4 applies.

```
1  rule {
2    include getTopItems from "socket://localhost:8001" with "soap"
3    include processRecommendations from "socket://localhost:8003" with "soap"
4
5    newRoles: R
6    on { N.tag == "recommender" and E.recommender == "low-power" }
7    do {
8        getPopularProducts: R() -> P();
9        items@P = getTopItems( 10, "popularity" );
10       popularProducts: P( items ) -> R( items );
11       recommendations@R = processRecommendations( items );
12       recommendedProducts: R( recommendations ) -> W( recommendations );
13       recommender@W = true
14   }
15 }
```

Listing 4: Rule for the *low-power* flavour of the Recommender service.

The rule in Listing 4 includes two external services that provide the functionalities `getTopItems` from `"socket://localhost:8001"` and `processRecommendations` from `"socket://localhost:8003"`, both using the soap protocol. Moreover, the rule introduces the new role `R` (acting as coordinator for the Recommender service) to the choreography. The rule activates when two conditions are simultaneously met: the `scope`'s `tag` equals `"recommender"` and the environment variable `recommender` is set to `"low-power"`. When triggered, the rule implements a low-power recommendation flow where the Recommender requests popular products from the Persistence service, which then calls the `getTopItems` function to retrieve the most popular items. Persistence sends these popular items back to the Recommender, which processes them using the `processRecommendations` function. The Recommender then sends the processed recommendations to the WebUI service, which sets its `recommender` flag to `true`, indicating that the recommendations are available.

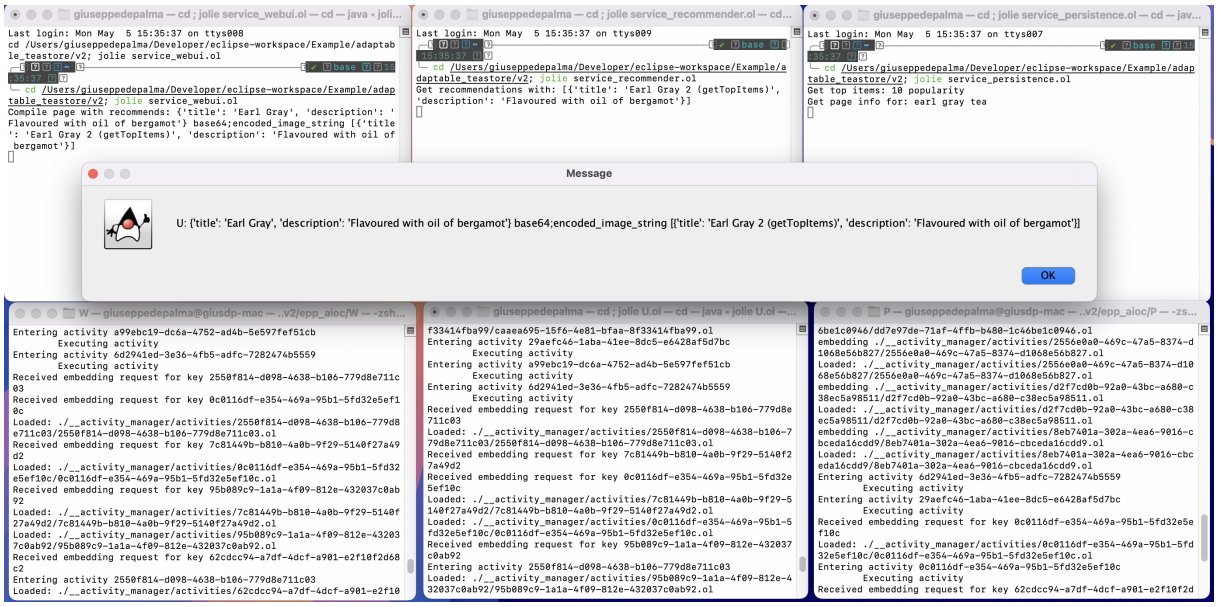

Figure 3: Execution of Barebone Adaptable TeaStore with the low-power Recommender version.

While the previous rule allows one to compute recommendation information in low-power mode, a second rule is needed to use this information at page compilation. The rule is described in Listing 5. Notably, the rule showcases a third option one can use in applicability conditions: local variables of the role in charge of managing the adaptation, as in `recommender == true`, where `recommender` is a variable of role `W` (webUI).

```
1 rule {
2   include compilePageWithRecommends from "socket://localhost:8000" with "soap"
3
4   on { N.tag == "page_compiler" and recommender == true }
5   do {
6     page@W = compilePageWithRecommends( info, img, recommendations );
7     recommender@W = false
8   }
9 }
```

Listing 5: Rule for the compilation of the page with recommendations.

As done in the previous section, we illustrate the execution of the adaptable choreography with the screenshot reported in Figure 3, which captures an execution step after the application of the rule for the adaptation with the low-power version of the Recommender. In particular, we notice that the resulting message to the user is similar to the one in Figure 2, although, in Figure 3, it integrates the suggestion of the Recommender.

On the Application of Adaptation Rules When, during execution, the runtime enters a scope, it checks all available rules for applicability (in no specific order) and it applies the first one whose condition evaluates to true. Specifically, the runtime performs the check for applicability in sequence, con-

sidering the rules one by one w.r.t. the current status of the system. Since environmental properties can change in parallel with rule evaluation, race conditions can occur where the applicability condition of a rule is negative at check time but becomes positive afterwards, leading to scenarios where the runtime applies no rule, even despite having complete coverage of all possible states[4].

This issue is not easily amendable due to the concurrent nature of distributed architectures. Attempting to fix the environment state during rule checking (e.g., by taking a snapshot or locking environmental variables) would be counterproductive, as it would cease to provide a valid representation of the actual system status and could lead to applying rules based on desynchronised states. Moreover, the possibility that the runtime applies a rule while the system status changes (potentially causing errors) is not fundamentally different from other distributed system failure modes. For instance, a choreography might successfully adapt and then have an external resource crash during execution, leading to similar error conditions. We argue that one might more effectively address these race conditions through error handling and recovery mechanisms rather than attempting to eliminate the temporal inconsistencies that are inherent to distributed systems. The challenge lies in designing adaptation strategies that are resilient to these timing issues.

5 Barebone TeaStore Choreography, with Adaptable Authentication

We now consider a more complex adaptation scenario, where the Authentication service is added. The idea is that, if the user can authenticate/is authenticated, the system can provide them with personalised products, images, and recommendations, based on their preferences.

However, we have a tricky point on our hands: authentication impacts the choreography at many points, including when services, such as the Recommender, are possibly added at runtime. To coordinate and follow a consistent behaviour, we exploit a variable `token`, managed by the WebUI, to keep track of whether authentication has already been performed or not (cf. Listing 6, line 4). We also add various scopes to enable adaptation.

```
1  aioc {
2    address@U = getInput( "Insert address" );
3    getPage: U( address ) -> W( address );
4    token@W = "none";
5    {
6      scope @W {
7        getPageInfo: W( address ) -> P( address );
8        info@P = getPageInfo( address );
9        getInfo: P( info ) -> W( info )
10     } prop { N.tag = "page_info" } roles { U }
11     |
12     scope @W {
13       getPageImg: W( address ) -> I( address );
14       img@I = getPageImg( address );
15       getImg: I( img ) -> W( img )
16     } prop { N.tag = "page_images" } roles { U }
17     |
```

[4]Consider rules that apply w.r.t. the state of a database connection, which can either be “primary”, “replica” or “offline”. Let us assume that we enter an adaptation scope, the current status is “primary”, and that the runtime proceeds to check the applicability of a rule that requires the state to be “replica”. The condition is negative, and the runtime moves on to check the next rules. Meanwhile, the state changes to “replica”, which prevents the application of the other two rules. The result is that no rule applies, even though the system has complete coverage for all the database states.

```
18      scope @W { skip } prop { N.tag = "recommender" } roles { U, P }
19    };
20    scope @W {
21      page@W = compilePage( info, img )
22    } prop { N.tag = "page_compiler" };
23    getPage: W( page ) -> U( page );
24  }
```

Listing 6: Adaptable TeaStore barebone version

The main rule providing authentication is described in Listing 7. Its applicability condition specifies that it applies to scopes `"auth"` and requires authentication support to be available (by checking `E.auth == "available"`). Note that the rule applies only if no token has been obtained yet. These conditions model the fact that authentication should be ideally performed once, providing a token that can be used by all the functionalities needing it.

```
1  rule {
2    include login from "socket://localhost:8004" with "soap"
3    newRoles: A
4    on { N.tag == "auth" and E.auth == "available" and token == "none" }
5    do {
6      credentials@U = getInput( "Insert Credentials" );
7      sendCredentials: U( credentials ) -> A( credentials );
8      token@A = login( credentials );
9      if ( token != "none" )@A {
10       sendToken: A( token ) -> W( token )
11     }
12   }
13 }
```

Listing 7: Rule for the Authentication service.

Listing 8 shows the rule for refined page info compilation. In practice, the service which requires authentication `getPageInfoAsLoggedUser` is used if either the user authenticated beforehand and the system has their token, or if the scope inside the rule at line 6 is updated, thus obtaining the token "on the fly".

The rule in Listing 8 shows that we can have nested adaptations by using scopes inside rules. Note that this rule is applied independently of the availability of authentication facilities (e.g., there is no condition `E.auth == "available"`). This is meaningful in a scenario where authentication facilities may appear and disappear during the computation, e.g., the user authenticated at a previous stage, the authentication facility is not available any more, but we can still use the token to process the user's requests.

```
1  rule {
2    include getPageInfo, getPageInfoAsLoggedUser from "socket://localhost:8001" with "soap"
3    on { N.tag == "page_info" }
4    do {
5      getPageInfo: W( address ) -> P( address );
6      scope @W { skip } prop { N.tag = "auth" } roles { U };
7      if ( token != "none" )@W {
8        sendToken: W( token ) -> P( token );
```

```
9         info@P = getPageInfoAsLoggedUser( address, token )
10      } else {
11        info@P = getPageInfo( address )
12      };
13      getInfo: P( info ) -> W( info )
14    }
15  }
```

Listing 8: Rule for the Persistent interaction to retrieve product info.

Listing 9 provides adaptation for the full-power version of the Recommender. The logic of this rule is similar to the one from Listing 8, where nested adaptation provides authentication when needed.

```
1  rule {
2    include getPageInfo, processQuery,
3            getPageInfoAsLoggedUser from "socket://localhost:8001" with "soap"
4    include getQuery, getQueryAsLoggedUser,
5            processRecommendations from "socket://localhost:8003" with "soap"
6    newRoles: R
7    on {
8      N.tag == "recommender" and E.recommender == "full-power"
9    }
10   do {
11     getPageInfo: W( address ) -> P( address );
12     scope @W { skip } prop { N.tag = "auth" } roles { U };
13     if ( token != "none" )@W {
14       sendToken: W( token ) -> P( token );
15       info@P = getPageInfoAsLoggedUser( address, token );
16       getInfo: P( info ) -> R( info );
17       sendToken: W( token ) -> R( token );
18       query@R = getQueryAsLoggedUser( info, token )
19     } else {
20       info@P = getPageInfo( address );
21       getInfo: P( info ) -> R( info );
22       query@R = getQuery( info )
23     };
24     sendQuery: R( query ) -> P( query );
25     result@P = processQuery( query );
26     queryResult: P( result ) -> R( result );
27     recommendations@R = processRecommendations( result );
28     recommendedProducts: R( recommendations ) -> W( recommendations );
29     recommender@W = true
30   }
31 }
```

Listing 9: Rule for the *full-power* flavour of the Recommender service.

Note that the scopes which can provide adaptation are in parallel in Listing 6. Hence, depending on the scheduling, multiple authentications may be performed. This behaviour happens, in particular, if the conditions for applicability of the authorisation rule are all checked before the variable token gets a "none" value. On the one hand, apart for the burden for the user to authenticate multiple times, no issues is caused. On the other hand, avoiding this would require to sequentialise the different steps, losing

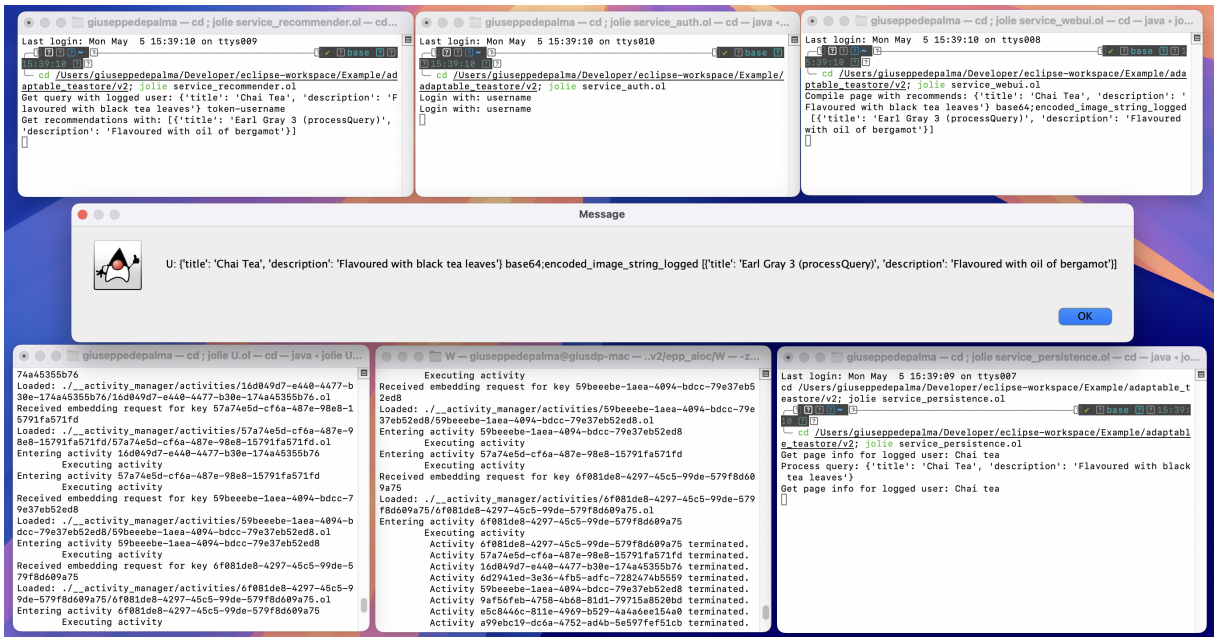

Figure 4: Execution of the Adaptable TeaStore choreography with the Auth service and the full-power Recommender version.

efficiency.

Closing the section, we report in Figure 4 a screenshot of the execution of the Adaptable TeaStore choreography after the full adaptation using the Auth service and the full-power Recommender. In the figure, we can see that the Auth service allowed the user to log in (visibile in the Auth service console; the login happens two times due to the parallel execution of the scopes in Listing 6). We also notice the execution of the full-power version of the Recommender from the result of the page compilation in the pop-up window.

6 Discussion and Conclusion

We see this contribution as a useful benchmark of the fitness of a language such as AIOCJ (and, to some extent, the choreographic programming approach) in capturing the concerns and possibilities of the TeaStore architecture and helpful to indicate and discuss implementation patterns and styles inspired by the choreographic paradigm.

Ephemeral Adaptations and Implicit Regressions Our approach with AIOCJ starts from a barebone system, where we strategically insert adaptation scopes at points where we anticipate potential adaptation needs. The application of adaptation rules relative to the available scopes and system environment states then adapts using the appropriate choreography fragments (found in rules) to replace these scopes. This methodology implies that, for each interaction flow among the participants (i.e., the execution of the base choreography), the system adapts according to available rules and current conditions.

A significant advantage of this approach is that we do not need to specify "regression" adaptation rules, because the system always starts executing from the original version and possibly adapts according

to the available rules and rule applicability conditions. This point becomes more practical if we consider "wrapping" the behaviour found within the `aioc` scope in the examples with a `while(e)@W{ ... }`, making the whole system loop (as long as the evaluation of expression `e` at `W` evaluates to true) and implementing in a more faithful way the behaviour of the Adaptable TeaStore — where the User can interact with (and request content from) the WebUI multiple times. Considering the full Adaptable TeaStore implementation (from Listing 6), using the `while` loop would mean that, at each iteration, the system would start from the "barebone" version and then adapt according to the current status of the system — e.g., during an iteration the Auth service might be unavailable, preventing the application of (parts of the) rules that concern the latter, while at the next one the service might become available, supporting the application of the related rules. From the language standpoint, similarly to `if`s, `while`s require the indication of a controller, i.e., a participant (in the example above, `W`) that determines whether the participants of the loop (including the controller) shall execute the code within the `while` scope or continue with the next instruction.

Generally, we can describe the application of AIOCJ adaptations as "ephemeral" (i.e., bound to a give flow iteration, but otherwise volatile w.r.t. the overall system status) which, complementarily, determines the implicit regression of the system to the original state.

While the ephemeral application of adaptation provides high flexibility, we notice that this pattern can introduce potential performance costs, since adaptation can occur multiple times (e.g., at each iteration and, separately, at different execution steps of the choreography), and "fluctuations" in user experience and system states, if rules are not applied deterministically. For instance, a user could initially successfully log in because the Auth service is available; then, due to the non-deterministic application of rules, the rule enabling the usage of the Auth service might not apply, and the user might experience an "inconsistent" behaviour, given their previous successful login. That said, in conventional systems, if the Auth service fails, users would typically encounter a standard error (e.g., HTTP 500), determining fluctuations similar to the one mentioned above. In general, we consider exploring the practical implications of applying the ephemeral adaptation pattern in these contexts an interesting future endeavours for refining the (approach behind the) AIOCJ language.

Integrating Service Controllers One important element we abstracted away in our modelling is that, at the AIOCJ level, we do not manage provisioning, scaling, or other infrastructure concerns. AIOCJ, at the moment, provides no structured way to specify or reason about these concerns, which we envision incorporating into choreographies in the future.

In practice, one could integrate the AIOCJ language and runtime with external controllers for service (de)allocation, such as Kubernetes. These controllers can react to various events like traffic increases or node failures by allocating services, relocating them, etc. The services they manage are the Adaptable TeaStore components interconnected through our choreography. Thus, one should have a way of integrating service controllers with choreographic implementations so that the availability of services, load balancing, and similar concerns are integrated within the choreographic approach.

Modelling Functional and Non-Functional Concerns in AIOCJ The discussion above, on integrating AIOCJ constructs with architecture controllers, introduces a more general issue, which is the separation of functional and non-functional concerns — the latter include performance, availability, scalability, resilience, and reliability. Specifically, while AIOCJ does not explicitly model these aspects, they can surreptitiously show up in adaptation code. Considering our examples, we adapt the architecture according to the presence/absence of certain components, as witnessed by the applicability con-

ditions `E.recommender == "low-power"` and `E.auth == "available"` found within the adaptation rules, which e.g., specify the handling of different service flavours as part of the functional specification.

Hence, at the moment, AIOCJ provides no structured way to specify or reason directly about these non-functional properties which, nonetheless, are relevant to adaptation. This mixing surreptitiously introduces a hybrid approach where some non-functional aspects are choreographically coordinated while others are delegated to external controllers, like Kubernetes, which further strengthens the point for integrating external controllers at the choreographic level.

Adaptation Compositional Complexity An alternative to the way we modelled the compositionality of Adaptable TeaStore scenarios and configurations involves creating rules for each service configuration, e.g., rules for "Auth + Recommender full-power", "Auth + Persistence", etc. The challenge with this approach is the need for numerous rules. Indeed, while this alternative pattern could, in principle, lead to more easy-to-interpret adaptation scenarios (there is no need to "figure out" what combination of rule applications one could obtain, depending on the availability of scopes and rules) it could lead to an "explosion" of adaptation rules, depending on the coupling between participant/service behaviours within the choreography. For this reason, in this paper, we opted for the nesting of rules, e.g., as seen in Listing 8 and Listing 9. Another interesting research direction is to explore in which contexts one pattern might be more suitable than the other, considering both qualitative traits, e.g., in terms of how easily programmers can specify the expected behaviour of an adaptable system, and in terms of performance, e.g., given that nested adaptations may involve more runtime steps than the "flat" ones.

Choreographic Service Composition As a general note, we observe that Adaptable TeaStore (like TeaStore) is highly orchestration-oriented, with WebUI orchestrating services provided by other components. In this paper, we faithfully model TeaStore's behaviour, implementing a choreography that follows an idiosyncratic orchestration-oriented composition pattern (where the WebUI centralises most interactions), future extensions of this work could propose alternative, idiomatic choreographic versions of the TeaStore, e.g., where we support direct communication between services without routing them through the WebUI, e.g., for increased efficiency.

We argue that the apparent mismatch between Adaptable TeaStore's architectural pattern (centralised orchestration) and the AIOCJ modelling approach (distributed choreography) helps us to demonstrate the flexibility of AIOCJ in handling non-idiomatic patterns, showing how choreographic approaches can refactor orchestration-oriented systems and that choreographic correctness guarantees (like deadlock freedom) remain valuable even when modelling centralised architectures.

Error handling While, for brevity, we did not discuss the issue of interacting with failing services from an AIOCJ choreography, we notice that the language gives little support service interaction handling.

Indeed, when a choreography communicates with an external service, if the latter fails, the entire choreography breaks down. This risk represents a fundamental design challenge in the AIOCJ model.

To prevent stopping the execution due to an external service failure, the AIOCJ runtime should include an intermediary service layer (that cannot fail and) capable of handling requests from the choreography and returning values indicating success or failure of the operation. Such a mediator would enable adaptation rules to be triggered based on service availability. For instance, when contacting the Auth service and receiving a response indicating the service is unreachable, the system could apply an appropriate adaptation rule specifically designed for this scenario. The current model lacks this intermediary resilience layer, creating a brittle dependency between choreographies and external services.

Alternatively, one might introduce choreography-level error handling mechanisms that directly trigger adaptation when external service failures occur. This extension would require that AIOCJ incorporates exception handling constructs that can seamlessly transition into adaptation scenarios.

Generalising, the discussion above exposes a broader issue with choreographic approaches like AIOCJ: while they excel at ensuring correctness properties during normal operation and adaptation transitions, they struggle with graceful degradation when facing unexpected external failures. The strong coupling between choreographic descriptions and external services creates a single point of failure that contradicts the resilience goals of modern Cloud architectures. Furthermore, this issue compounds when adaptation rules themselves depend on potentially failing external services. In such cases, the very mechanism designed to handle changing requirements becomes vulnerable to the same failure modes it aims to address.

Considering the specific case of Adaptable TeaStore, the aspect of error handling becomes particularly evident when considering the architecture's multiple service variants and failover scenarios. While AIOCJ can express the transitions between service configurations, it struggles to handle the detection and management of the failures that would require such transitions in the first place.

A strategy towards supporting error handling in AIOCJ is having monitors that track the state of external services and feed this information to the environment. When external services go down, the environment becomes aware of this status change and can trigger appropriate adaptations. However, this monitoring approach has inherent limitations. It only works effectively if the monitor has an updated status of the external service when the scope begins and when the adaptation check is performed. More critically, this approach provides no protection if the external service breaks during the execution of the scope, after the adaptation check has already occurred. This timing vulnerability creates a window of failure that cannot be addressed through conventional adaptation mechanisms in AIOCJ, highlighting the need for more robust error handling capabilities integrated directly into the choreographic model.

Adaptation flows Another interesting refinement point for the AIOCJ language emerges from the usage of bookkeeping variables to coordinate the behaviour of different, adaptable parts of the choreography. Indeed, in our AIOCJ model of the Adaptable TeaStore, we introduced several variables to track the state and availability of services across adaptation scenarios. For instance, we needed variables to record whether the Auth service was available and which version of the Recommender was active. These variables are not part of the conceptual model of the TeaStore but represent implementation artefacts required to bridge the gap between AIOCJ's adaptation mechanisms and the actual system state.

The use of these bookkeeping variables introduces several problems, such as making the choreography description more complex and harder to understand, as readers must mentally track the state of these variables alongside the actual business logic. Moreover, bookkeeping creates potential for inconsistencies, as variables might not be properly updated in all execution paths, making also proposing static verification more difficult, as formal analysis must account for these additional state elements.

To address these limitations, refinements to the AIOCJ language could introduce direct support for adaptation based on service states (e.g., integrating runtime interaction of the rule with orchestrators, such as Kubernetes), which would make adaptation conditions more declarative and closely aligned with the conceptual model of the system.

References

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

[2] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese & Jacopo Mauro (2017): *Dynamic Choreographies: Theory And Implementation*. *Log. Methods Comput. Sci.* 13(2), doi:10.23638/LMCS-13(2:1)2017. Available at https://doi.org/10.23638/LMCS-13(2:1)2017.

[3] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro & Maurizio Gabbrielli (2014): *AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications*. In Benoît Combemale, David J. Pearce, Olivier Barais & Jurgen J. Vinju, editors: *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, *Lecture Notes in Computer Science* 8706, Springer, pp. 161–170, doi:10.1007/978-3-319-11245-9_9. Available at https://doi.org/10.1007/978-3-319-11245-9_9.

[4] Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro & Maurizio Gabbrielli (2017): *Programming Adaptive Microservice Applications: An AIOCJ Tutorial*. In Simon Gay & António Ravara, editors: *Behavioural Types: from Theory to Tools*, River Publishers, pp. 147–167, doi:10.13052/rp-9788793519817. Available at https://doi.org/10.13052/rp-9788793519817.

[5] Saverio Giallorenzo, Ivan Lanese & Daniel Russo (2018): *ChIP: A Choreographic Integration Process*. In Hervé Panetto, Christophe Debruyne, Henderik A. Proper, Claudio Agostino Ardagna, Dumitru Roman & Robert Meersman, editors: *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*, *Lecture Notes in Computer Science* 11230, Springer, pp. 22–40, doi:10.1007/978-3-030-02671-4_2.

[6] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira & Gianluigi Zavattaro (2016): *Foundations of Session Types and Behavioural Contracts*. *ACM Comput. Surv.* 49(1), pp. 3:1–3:36, doi:10.1145/2873052. Available at https://doi.org/10.1145/2873052.

[7] Jóakim 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: *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2018, Milwaukee, WI, USA, September 25-28, 2018*, IEEE Computer Society, pp. 223–236, doi:10.1109/MASCOTS.2018.00030. Available at https://doi.org/10.1109/MASCOTS.2018.00030.

[8] Fabrizio Montesi (2023): *Introduction to Choreographies*. Cambridge University Press, doi:10.1017/9781108981491.

[9] Fabrizio Montesi, Claudio Guidi & Gianluigi Zavattaro (2014): *Service-Oriented Programming with Jolie*. In Athman Bouguettaya, Quan Z. Sheng & Florian Daniel, editors: *Web Services Foundations*, Springer, pp. 81–107, doi:10.1007/978-1-4614-7518-7_4. Available at https://doi.org/10.1007/978-1-4614-7518-7_4.