

# A Conceptual Framework for API Refactoring in Enterprise Application Architectures

Fabrizio Montesi<sup>1</sup>, Marco Peressotti<sup>1</sup>, Valentino Picotti<sup>1</sup>, and Olaf Zimmermann<sup>2</sup>

<sup>1</sup> Department of Mathematics and Computer Science, University of Southern Denmark {fmontesi,peressotti,picotti}@imada.sdu.dk

<sup>2</sup> University of Applied Sciences of Eastern Switzerland olaf.zimmermann@ost.ch

**Abstract.** Enterprise applications are often built as service-oriented architectures, where the individual services are designed to perform specific functions and interact with each other by means of well-defined APIs (Application Programming Interfaces). The architecture of an enterprise application evolves over time, in order to adapt to changing business requirements. This evolution might require changes to the APIs offered by services, which can be achieved through appropriate API refactorings. Previous studies on API refactoring focused on the effects on API definitions, with general considerations on related forces and smells. So far, instead, the development strategy for realising these refactorings has received little attention. This paper addresses exactly this aspect.

We introduce a conceptual framework for the implementation of API refactorings. Our framework elicits that there are important trade-offs and choices, which significantly affect the efficiency, maintainability, and isolation properties of the resulting architecture. We validate our framework by implementing several refactorings that introduce established API patterns with different choices, which illustrates the guiding principles offered by our framework. Our work also elicits, for the first time, how certain programming language features can reduce friction in applying API refactoring and open up more architectural choices.

**Keywords:** Enterprise application · Microservices · API Refactoring

## 1 Introduction

Enterprise applications are distributed information systems designed to support the needs of complex organisations, for example by managing business processes and handling data [10]. Such applications are often built using (micro)service-oriented architectures, where individual services perform specific functions and interact through well-defined APIs (Application Programming Interfaces) [6].

As organisations evolve, so do their applications: enterprise application architectures must continuously adapt to changing business requirements [7–9]. In the context of APIs, this prompted studying *API refactoring*: the modification of interfaces to improve quality attributes, such as efficiency [24, 25].

Recently, a catalogue of API patterns provided a basis for API refactorings [25,31]. Previous studies mainly focused on the high-level forces and smells, like modifiability and high latency, that motivate and guide API refactorings. Conversely, it is yet unknown how developers are supposed to implement an API refactoring and assess its quality; a research gap that we address in this work.

In this paper, we introduce EMI (efficiency, maintainability and isolation), a conceptual framework for assessing the implementation of API refactoring in service architectures. EMI centres around two dimensions: 1) *generality*, which assesses the degree of abstraction of the refactored API source code; 2) *distribution*, which elicits where the refactored API source code resides. Realising the combination of both dimensions results in six development strategies, each representing design choices for the implementation with respective trade-offs. The trade-offs pertain the quality aspects of *efficiency* (E), *maintainability* (M), and *isolation* (I) of the resulting architecture. We score each of these three aspects from 1 to 3 for our strategies, yielding the EMI score for API refactoring. Clearly, there is no silver bullet: no strategy scores perfectly (9), emphasising the importance of making conscious implementation decisions.

To validate our framework in practice, we apply it to several API refactorings implemented in the service-oriented programming language Jolie [19]. We choose Jolie because of its abstraction and expressiveness capabilities. Firstly, Jolie offers a technology-agnostic language for defining APIs and native constructs for declaring endpoints that consume and provide APIs. This makes our refactorings direct and easy to understand. Secondly, Jolie offers features not present in other programming languages, such as API polymorphism and aggregation (the merging of endpoints), allowing us to implement design choices declaratively.

In more detail, we apply our six development strategies to the same refactoring: the introduction of the API KEY pattern – which rejects requests without a valid key (Section 4.1) – to a service that offers a catalogue of scientific publications. We then broaden our study to patterns that do not require behavioural changes, MERGE ENDPOINTS and VERSION IDENTIFIER, reaching modular solutions (Section 4.2). A main finding is that our framework can be used to distill systematic recipes for API refactoring, which developers can mechanically apply step by step to achieve an implementation with a declared EMI score (Section 5).

In summary, we contribute the EMI framework, a scheme to assess the adaptation of API refactorings, we validate EMI by applying API refactorings to (micro)service architectures, and provide canonical recipes to obtain certain EMI scores for the architecture evolution. Thus, our work offers developers the possibility to assess and implement API refactorings in practice and lays the grounds to explore the impact of refactoring APIs in further detail.

## 2 Related Work

Our study builds on the reference catalogue of patterns for API design [31], which addresses the challenge of remote API design [29] through peer-reviewed patterns published in the period 2017–2020 [17,26–28,30].

API refactoring was previously investigated based on the same catalogue [24, 25]. Those studies focus on architectural considerations and especially *why* and *when* an API pattern should be introduced, considering forces and smells. Differently, our work is the first to investigate *how* an API pattern is implemented. Specifically, our interest lies in the different choices that one can make regarding the code of a refactoring, and the quality trade-offs that they yield. Another difference is that we study how to refactor also the implementation of an API, whereas previous work discusses only how to refactor the API definition. Our frameworks can be seen as a refinement of Attribute-Driven Design and Architecture Trade-off Analysis Method which are concerned with informing and assessing architectural decisions in light of quality attribute requirements [2].

Jolie is an emerging programming language for service-oriented computing [19], which has recently been gaining traction because of its native features for defining and composing services [1, 11–13, 15, 16]. Previous work has validated Jolie in different domains, including system integration [11, 18], Internet of Things [11, 16], journalism [5], and cloud provisioning [14].

The work nearest to ours is perhaps the implementation of an API- and location-agnostic circuit breaker [20] – a pattern for increasing resilience [21]. In that development, Jolie’s high-level abstractions are used to obtain a flexible implementation and experiment with different deployment strategies (in the client, in a forwarding proxy, or in the server). These have important security implications [20] that were later confirmed in an independent study [4]. The developments in [20] fall under the Parametric/Adjacent and Parametric/External strategies in our framework (see Section 3). Our interest in the present work is much more general: rather than focusing on a specific use case, we formulate a framework that can be used to reason about any API refactoring. Furthermore, the quality aspects considered here are not considered in [20].

Many frameworks and languages for service-oriented systems have been proposed, including [Spring Boot](#), [Express](#) for Node.js, [Ballerina](#) [23], and [WS-BPEL](#) [22]. Some of Jolie’s features that we use in our work can be found or implemented in these technologies. Aggregation and redirection – Jolie for merging and redirecting endpoints – recall the routing mechanism in [Express](#). Jolie uses structural typing, like [Ballerina](#) [23], which is more suitable to networked (and multi-technology) systems than nominal typing. On top of offering all the features we need in a single package, Jolie’s primitives are designed to make the resulting APIs statically known, which is useful in the context of API design.

### 3 The EMI Framework for API Refactoring

We now present our conceptual framework for API refactoring – the EMI framework. It is depicted in Table 1. We explain it in the rest of this section.

API refactoring changes both an interface and its implementation, while improving at least one quality attribute [24, 25]. This may affect the external behaviour of an API observed by clients, without altering its capabilities.

		Distribution		
		Internal	Adjacent	External
Generality	Parametric	E ★★★☆	E ★★★☆	E ★☆☆☆
		M ★★★☆	M ★★★★★	M ★★★★★
		I ★☆☆☆	I ★★★☆	I ★★★★★
	Ad-hoc	E ★★★★★	E ★★★☆	E ★☆☆☆
		M ★☆☆☆	M ★★☆☆	M ★★☆☆
		I ★☆☆☆	I ★★☆☆	I ★★★★★

**Table 1.** The EMI framework.

We introduce some terminology. In the remainder, we refer to the changes introduced by an API refactoring as the *new functionality*, bearing in mind that such functionality does not alter the feature set of the API [25]. In line with the API domain model of [31], we consider an API to be a collection of operations that can be invoked by clients. Services can offer APIs through *endpoints*, which expose operations at a designated location according to a given transport protocol. We call such services *API providers*. We distinguish the API and implementation that we start from and then end up with after a refactoring with the prefixes *original* and *refactored*.

### 3.1 Generality and distribution

The EMI framework focuses on two dimensions to assess the quality attributes of the implementation of an API refactoring: *generality* and *distribution*.

The *generality* dimension concerns whether the implementation of the new functionality depends on or abstracts from the definition of the original API. We identify two possibilities.

**Ad-hoc** The code of the new functionality depends on hardcoded information on the names, types, or behaviours of the operations in the original API.

**Parametric** The code of the new functionality abstracts from the names, types, or behaviours of the operations in the original API.

Generality serves as an indicator of the logical coupling between the new code and the old. It is significant because API patterns provide, at least conceptually, reusable solutions to recurring problems. Thus, in a way, generality indicates how much this reusability is achieved in real code.

The *distribution* dimension concerns where the code for the new functionality is located in relation to the original API provider and its clients. There are three possibilities, depicted in Fig. 1.

**Internal** The code of the new functionality is mixed with the code of the original API provider. Thus, they share state. After the refactoring, the original API provider becomes the refactored API provider.

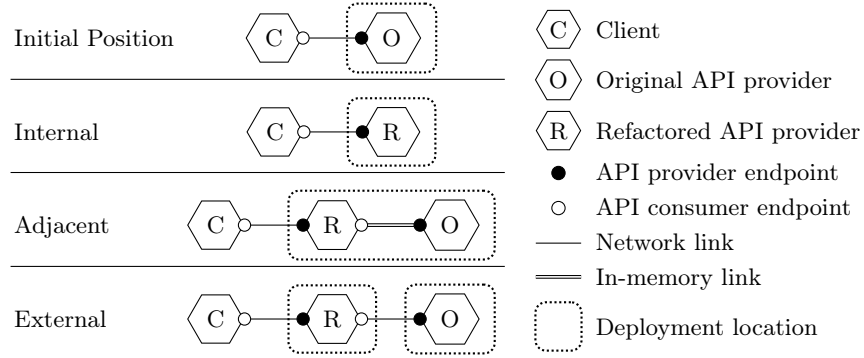


Fig. 1. Possible choices for distribution.

**Adjacent** The refactored API provider is a separate service. They have separate state and are executed independently, but they are deployed such that they can communicate efficiently through local resources (local memory channels, inter-process communication, loopback network interfaces, etc.).

**External** The refactored API provider is a separate service. It is deployed remotely from the original, and thus can communicate with it only through network communication.

### 3.2 EMI scores

The combination of the axes of generality and distribution gives rise to six possible development strategies, each presenting different trade-offs. To help in navigating these trade-offs, we score each strategy on three quality attributes using a three-level scale (★☆☆, ★★☆☆, or ★★★): efficiency (E), maintainability (M), and isolation (I). We explain each score next.

#### Efficiency (E)

- E ★★★** The new functionality is implemented optimally, with no extra overhead caused by design choices.
- E ★★☆☆** Design choices cause extra overhead in terms of local resources (memory, local communication, etc.).
- E ★☆☆☆** Design choices cause extra overhead in terms of remote resources (e.g., network communication).

#### Maintainability (M)

- M ★★★** The original and refactored API providers can be maintained independently.
- M ★★☆☆** The implementations of the new functionality and the original API provider are separate but tightly coupled.
- M ★☆☆☆** The implementations of the new functionality and the original API provider are completely mixed.

#### Isolation (I)

- ! ★★★ The original and refactored API providers do not share any local resources for their execution.
- !★★☆ The original and refactored API providers share execution resources (e.g., CPUs, memory), but do not share state and interact purely by means of the original API.
- !★☆☆ The new functionality and the original API implementation share internal program state (e.g., stack, variables, heap).

The levels of these scales are intentionally broad, in order to avoid being tied up by very specific technological details. This is in line with the technology-agnosticism of microservices [6].

### 3.3 Scoring development strategies

We end the presentation of our framework with an analysis that justifies the EMI scores of each development strategy, referring also to examples of API patterns and technologies where relevant.

**Ad-hoc/Internal (E ★★★ M★☆☆ I ★☆☆)** This is the most efficient strategy, because the new functionality is implemented directly by changing the behaviour of the original API provider. Thus, the code of the new functionality encounters no unnecessary overhead in integrating with the original implementation. For example, introducing the `PAGINATION` pattern to an implementation that queries a database gives the possibility to modify the query in order to retrieve fewer results – those for the page being requested. For the very same reasons, however, this is also the least maintainable and isolated choice, since the new code is mixed and shares all resources with the old code. Examples of this strategy are shown in Sections 4.1 and 5.

**Parametric/Internal (E★★☆ M★★☆ I★☆☆)** This strategy trades some efficiency for maintainability by abstracting from the operation names and behaviours of the original API. The code of the new functionality can be reused across different APIs, but has limited access to changing their behaviour: the new functionality can only intercept, modify, and conditionally forward request and response message to and from the original implementation. Examples of this strategy are implementations adopting [Java Servlet Filters](#) or [Express middleware functions](#).

**Ad-hoc/Adjacent (E★★☆ M★★☆ I★★☆)** Compared to Ad-hoc/Internal, implementing the new functionality in a separate component trades some efficiency for partially improved maintainability and isolation. However, the new functionality remains coupled with the original API (ad-hoc), so changes to the original API require updating the refactored API provider, too. Thus, maintainability is still not ideal. Improved isolation comes at the cost of some overhead in the interaction between the refactored and original API providers. Efficiency is further affected by the new functionality not having access to changing the internal behaviour of the original API provider. This strategy can be implemented with, for example, the [sidecar pattern](#), the [ambassador pattern](#), or Jolie’s embedded services (see Section 4.1).

**Parametric/Adjacent (E★★☆ M★★★★ I★★☆)** This strategy has the same efficiency and isolation characteristics as the previous one, but greatly improved maintainability by decoupling the implementation of the new functionality from the operation names and message types of the original API. The sidecar and ambassador patterns are again useful to implement this strategy. Jolie’s embedded services combined with couriers and interface extenders (see Section 4.1) offer an interesting solution, because the refactored API can be automatically and statically computed.

**Ad-hoc/External (E☆☆ M★★☆ I★★★★)** The strategy with the highest level of isolation, since the new functionality interacts with the original API provider only via remote access. For the same reason, it is also the least efficient strategy. This strategy does not achieve the highest maintainability score due to the coupling between the new functionality and the original API. This strategy can be implemented simply by developing a proxy service offering the refactored API and forwarding each operation invocation to the original API provider when appropriate.

**Parametric/External (E☆☆ M★★★★ I★★★★)** This strategy has the same efficiency and isolation scores as the previous one, but also the highest maintainability score for the same reason given for Parametric/Adjacent.

No strategy scores a perfect 9. The reason lies in the unavoidable tension between efficiency and isolation: optimal efficiency requires sharing resources, which prevents achieving optimal isolation.

## 4 Validation

In this section, we validate our framework by applying it in depth – exploring all our strategies for a single pattern – and in breadth – applying selected strategies to other patterns.

### 4.1 API Key in Jolie

We illustrate the use of our framework by applying each strategy to a concrete use case: the introduction of the API KEY pattern to a service managing a catalogue of scientific publications. API KEY identifies clients through respective unique keys, which must be included in requests.

We code our examples in Jolie. In Jolie, the operations and message types of an API are defined as an **interface**. The next interface defines the API of our publication catalogue service.

```

1  type Publications: { publications*: Publication }
2  type Publication: Proceeding | InProceeding | Article
3
4  interface PubCatInterface {
5    RequestResponse:
6      getAuthorPubs( {authorId: string} )( Publications )
7      getConfPubs( {confId: string} )( Publications )
8  }
```

PubCatInterface comprises two operations: `getAuthorPubs`, which expects the unique identifier of an author (as the field `authorId` of the request message) and returns all their publications (message type `Publications`); and `getConfPubs`, which given a conference identifier (`confId`) returns the publications of that conference. The type `Publications` describes a record with a field `publications` containing zero or more (\*) values of type `Publication`. `Publication` is the union of three types (omitted) corresponding to proceedings (`Proceeding`), papers in proceedings (`InProceeding`), and journal articles (`Article`).

Interfaces are offered to clients by defining an **inputPort**, Jolie for an endpoint that accepts remote invocations. An input port is defined inside of its enclosing **service** and commits to a concrete **location** and transport **protocol** (HTTP, SOAP, binary protocols, etc.). The definition of our publication catalogue service is given next (abstracting some internal implementation details).

**Listing 1.** Original API Provider.

```

1  /* Service definition */
2  service PubCat {
3    /* API Endpoint */
4    inputPort ip {
5      location: "socket://localhost:8080"
6      protocol: http { format = "json" }
7      interfaces: PubCatInterface
8    }
9    /* Behaviour */
10   main {
11     [ getAuthorPubs( request )( response ) { /* fetch the data from db */ } ]
12     [ getConfPubs( request )( response ) { /* fetch the data from db */ } ]
13   }
14 }

```

In Lines 4 to 8, service `PubCat` exposes `PubCatInterface` on TCP port 8080 over the HTTP protocol with message payloads in JSON format. Its implementation (Lines 10 to 13) consists of an *input choice* that can react to any invocation of the operations it lists. Each branch in the choice has the form:

$$[ \text{operation}( \text{request} )( \text{response} ) \{ B \} ]$$

where `operation` is the name of the operation, `request` and `response` are the input and output parameters, and `B` is the code block computing the response.

Introducing the API KEY pattern requires extending request message types with an additional field `apiKey` (storing the key as a string) and declaring a faulty response message `NotAuthorised` for invocations with invalid keys. The refactored API is given next.

**Listing 2.** Refactored API.

```

1  interface PubCatInterfaceWithAPIKey {
2    RequestResponse:
3    getAuthorPubs( {authorId: string, apiKey: string} )( Publications )
4      throws NotAuthorised
5    getConfPubs( {confId: string, apiKey: string} )( Publications )
6      throws NotAuthorised
7  }

```



The refactoring of service PubCat and its interface PubCatInterface to obtain a service exposing the refactored API PubCatInterfaceWithAPIKey can be accomplished following any of the strategies outlined in Section 3. We illustrate their application and elicit the different features of Jolie that come to aid.

*Ad-hoc/Internal* We directly modify the code of both the original interface PubCatInterface and the service PubCat. PubCatInterface becomes the refactored API PubCatInterfaceWithAPIKey above. In PubCat, instead, the implementation of each operation is edited to validate the API key in the request message.

```

1  service PubCat {
2    inputPort ip { ... interfaces: PubCatInterfaceWithAPIKey }
3    main {
4      [ getAuthorPubs( request )( response ) {
5        /* check validity of request.apiKey */
6        if( isKeyValid ) { /* fetch the data from db */ }
7        else { throw NotAuthorised( /* fault data */ ) }
8      } ]
9      [ getConfPubs( request )( response ){ /* as for getAuthorPubs */ } ]
10   }
11 }

```

*Ad-hoc/External* We introduce a new service, PubCatWithAPIKey, with an endpoint exposing the interface PubCatInterfaceWithAPIKey. This service acts as an adapter for the original API provider, PubCat, which remains unchanged. The implementation of the API KEY pattern is entirely confined to the new service, which forwards valid invocations to PubCat. This requires the service PubCatWithAPIKey to declare an *output port* (Line 2) pointing to the API endpoint of PubCat. Its implementation (Lines 4 to 13) consists of an input choice where each operation checks the validity of the key in the request (`request.apiKey`). If the key is valid, then the key is erased from request (Line 8) before invoking the original operation `getAuthorPubs@pc` to obtain the intended response. Otherwise, the service replies with a faulty `NotAuthorised` message. Although the implementations of refactored and original API providers are separate, they must be kept in sync wrt future changes to the API, resulting in a negative impact to maintainability.

**Listing 3.** Ad-hoc/External refactored API provider.

```

1  service PubCatWithAPIKey {
2    outputPort pc { /* PubCat endpoint */ }
3    inputPort ip { /* ... */ interfaces: PubCatInterfaceWithAPIKey }
4    main {
5      [ getAuthorPubs( request )( response ) {
6        /* check validity of request.apiKey */
7        if( isKeyValid ) {
8          undef( request.apiKey ) /* remove API key before forwarding */
9          getAuthorPubs@pc( request )( response ) /* forward call */
10       } else { throw NotAuthorised( /* fault data */ ) }
11     } ]
12     [ getConfPubs( request )( response ) { /* as for getAuthorPubs */ } ]
13   }

```

*Ad-hoc/Adjacent* Jolie supports running separate services in the same application with its native **embedded** primitive. Thus, this strategy closely resembles the previous one, with the only difference being the deployment configuration of the two services `PubCat` and `PubCatWithAPIKey`. First, the service `PubCat` is promoted to an in-memory service by changing its location to `"local"` (Listing 1, Line 5). Then, we make the refactored API provider, `PubCatWithAPIKey`, embed the original `PubCat`. This is achieved by the statement `embed PubCat as pc` (Line 2), which instructs the Jolie runtime to load the service `PubCat` alongside `PubCatWithAPIKey` and make it reachable via an in-memory channel through the output port `pc`. These linguistic features allow for easily switching Jolie codebases between the *Adjacent* and *External* columns of the EMI framework, changing the deployment strategy based on performance considerations (*i.e.*, trade network overhead for CPU and memory consumption). However, in this strategy, the two interfaces `PubCatInterface` and `PubCatInterfaceWithAPIKey` are still separate definitions that need to be manually kept in sync.

**Listing 4.** Ad-hoc/Adjacent refactored API provider.

```

1 service PubCatWithAPIKey {
2   embed PubCat as pc
3   inputPort ip { /* Same as in Listing 1 */ }
4   main { /* Same as in Listing 3 */ }
5 }

```

*Parametric/Adjacent* To eliminate the coupling between refactored and original API providers, we leverage the Jolie language construct of an *interface extender*, which uniformly extends the types of all operations in an API. The extender `APIKeyExtender` defined in Listing 5 adds the `apiKey` field to all (\*) request messages and `NotAuthorised` as a new potential faulty response. `APIKeyExtender` precisely describes the changes we have to apply to `PubCatInterface` in order to obtain `PubCatInterfaceWithAPIKey`.

**Listing 5.** Parametric/Adjacent refactored API provider.

```

1 interface extender APIKeyExtender {
2   RequestResponse: *( {apiKey:string} )( void ) throws NotAuthorised
3 }
4
5 service PubCatWithAPIKey {
6   embed PubCat as pc
7   inputPort ip { /* ... */ aggregates: pc with APIKeyExtender }
8   courier ip {
9     [ interface PubCatInterface( request )( response ) {
10      /* check validity of request.apiKey */
11      if( isKeyValid ) { forward( request )( response ) }
12      else { throw NotAuthorised( /* fault data */ ) }
13    } ]
14 }
15 }

```

We use the interface extender to define the refactored API provider, service `PubCatWithAPIKey` (Lines 5 to 15). The service embeds `PubCat` (Line 6) and refers to it through output port `pc` as in Listing 4 above. Differently, however, input

port `ip` now **aggregates** `pc` with `APIKeyExtender` (Line 7), which instructs Jolie to forward messages for the API of `pc`, extended with `APIKeyExtender`, to `pc`.

Messages forwarded by means of aggregation (applications of **aggregates**) can be intercepted by means of a **courier** block. A courier is a piece of code attached to an input port, which gets executed whenever one of the input port's operations is invoked. The courier at Lines 8 to 14 implements the API KEY pattern for all operations of the interface `PubCatInterface`. Unlike a regular input choice, a courier can be parametric over the operation names of an interface:

```
[ interface PubCatInterface( request )( response ){ B } ]
```

where *B* is the code that is executed on each invocation of an operation of `PubCatInterface` on input port `ip`, and which can then decide whether to **forward** the request to the `PubCat` service, or return the error message `NotAuthorised`. The **forward** primitive automatically removes fields added by any interface extenders, so messages to `pc` are well-typed.

*Parametric/External* This solution differs from the previous one only on the deployment configuration of the refactored and original API providers, each having their own remote endpoint. The output port of Listing 5 must now describe the remote API endpoint of `PubCat`.

```
1 service PubCatWithAPIKey {
2   outputPort pc { /* PubCat endpoint */ }
3   inputPort ip { /* Same as in Listing 5 */ }
4   courier ip { /* Same as in Listing 5 */ }
5 }
```

*Parametric/Internal* This strategy can be easily expressed in Jolie's syntax by adding an interface extender and courier to the original API provider.

```
1 service PubCat {
2   inputPort ip { /* ... */ interfaces: PubCatInterface with APIKeyExtender }
3   courier ip { /* Same as in Listing 5 */ }
4   main { /* Same as in Listing 1 */ }
5 }
```

This is the only strategy that we could not test in Jolie, since its current interpreter does not support applying extenders to an interface that is not aggregated. This limitation is an implementation detail. Our study motivates the inclusion of this feature in future versions.

## 4.2 Other patterns: MERGE ENDPOINTS and VERSION IDENTIFIER

We now illustrate how to introduce two other patterns: MERGE ENDPOINTS and VERSION IDENTIFIER. Differently from API KEY, these patterns are fully architectural, in the sense that they do not introduce behavioural changes but rather affect only how APIs are accessed. We apply the Parametric/External strategy for both cases.

MERGE ENDPOINTS exposes the operations of two endpoints through a single endpoint. Suppose, for example, that we have a PubCat service for a publication catalogue and a CitInd service for citation indexing. We develop a new service, PublicationIndex, that merges their APIs by using aggregation.

```

1  service PublicationIndex {
2    outputPort pc { // publication catalogue
3      location: /* ... */ protocol: /* ... */
4      interfaces: PubCatInterface
5    }
6    outputPort ci { // citation index
7      location: /* ... */ protocol: /* ... */
8      interfaces: CitIndInterface
9    }
10   inputPort ip {
11     location: /* ... */ protocol: /* ... */
12     aggregates: pc, ic
13   } }

```

Note that aggregation requires the operations of the aggregated ports to have distinct names, which is in line with the pattern here. If this is not the case, one can use the other Jolie feature of redirection, explained in the next case.

VERSION IDENTIFIER exposes two (or more) different versions of the same API under a single endpoint. Here aggregation does not work, because the operation names in two versions of the same API likely overlap. Jolie solves this problem by offering the APIs under different *resource paths* at the same physical endpoint. In the next example, input port ip offers PubCatInterfaceV1 under path v1 and PubCatInterfaceV2 under path v2. Assuming that a client reaches the refactored API provider at location pubcat.com, this means that version 1 will be accessible at location pubcat.com/v1 and version 2 at location pubcat.com/v2.

```

1  service PubCatWithAPIKey {
2    outputPort pcv1 {
3      location: /* ... */ protocol: /* ... */
4      interfaces: PubCatInterfaceV1
5    }
6    outputPort pcv2 {
7      location: /* ... */ protocol: /* ... */
8      interfaces: PubCatInterfaceV2
9    }
10   inputPort ip {
11     location: /* ... */ protocol: /* ... */
12     redirects: v1 => pc1, v2 => pc2
13   }
14 }

```

This approach does not alter the original (versions of) the APIs, by distinguishing between versions based on the accessed location. Therefore, clients just need to be connected to the right location. An alternative to this approach is to extend the request types of all operations with a version identifier. However, this would require updating the clients to include this information. Furthermore, response types would become less precise, since they would need to accommodate the possible responses across all versions.

## 5 API Refactoring Recipes

In this section, we illustrate how our framework can be used to distill recipes that can be followed mechanically by programmers to apply an API refactoring. We cover the cases of a parametric implementation of the API KEY pattern and an ad-hoc implementation of the PAGINATION pattern. The latter is representative of situations where obtaining efficiency requires big sacrifices in maintainability and isolation.

We start with our recipe for API KEY.

---

### Refactoring recipe: Introduce API KEY (Parametric)

*Intent.* Introduce the API KEY pattern by means of a dedicated service that is parametric on the original API.

*Participants and Preconditions.*

1. Participant: A Jolie *service*, say `Original`, exposing the API subject to refactoring as an interface, say `OriginalAPI`.
2. Precondition: `Original` offers `OriginalAPI` through an *input port* `OriginalInputPort`.

*Refactoring steps.*

1. Introduce an *interface extender* `APIKeyExtender` that:
  - (a) Extends the request message with a field `apiKey` holding an API Key.
  - (b) Adds a faulty response message `NotAuthorised`.
2. Introduce a new *service* `OriginalWithAPIKey`:
  - (a) Introduce a new *output port* `original`.
  - (b) Choose between:
    - Choice 1 (External):** Configure output port `original` (at `OriginalWithAPIKey`) and input port `OriginalInputPort` (at `Original`) so that they communicate via the network.
    - Choice 2 (Adjacent):** Configure output port `original` (at `OriginalWithAPIKey`) and input port `OriginalInputPort` (at `Original`) so that they communicate via local memory.
  - (c) Introduce an *input port* `ip` that aggregates the output port `original` and extends it with `APIKeyExtender`.
  - (d) Introduce a *courier* for `ip` that intercepts all operations of `OriginalAPI` and:
    - i. Checks the validity of the API Key.
    - ii. If the key is valid, forwards the request to `original`.
    - iii. Otherwise, if the key is invalid, replies with the `NotAuthorised` response.

*Postconditions.*

1. Invoking any operation `op` at `OriginalWithAPIKey` with a valid `API Key` results into the same response message as the invocation to `op` at `Original` without an `API Key`.
2. Invoking any operation `op` at service `OriginalWithAPIKey` with an invalid `API Key` results into an `NotAuthorised` message.
3. Service `OriginalWithAPIKey` becomes the only client of service `Original`.

*Discussion and EMI scoring.* This recipe yields a parametric implementation, giving maintainability score `M★★★★`. Choice 1 introduces network overhead, giving `E★☆☆` and `I★★★★`, while Choice 2 does not, yielding `E★★★☆☆` and `I★★★☆☆`. We get the following possible EMI scores:

**Choice 1 (External):** `E★☆☆ M★★★★ I★★★★`.

**Choice 2 (Adjacent):** `E★★★☆☆ M★★★★ I★★★☆☆`.

---

We now present a recipe for the `PAGINATION` pattern. `PAGINATION` allows clients to retrieve smaller portions (‘pages’) of large data sets. The aim is to improve network and memory utilisation; this also addresses the stability antipattern of providing responses of unbounded size [21]. There are four variants of this pattern, corresponding to four different ways of identifying the page that the client wants [30,31]. Here, we implement the offset-based version.

---

### Refactoring recipe: Introduce `PAGINATION` (Ad-hoc/Internal)

*Intent.* Introduce the offset-based `PAGINATION` pattern for an operation.

*Participants and Preconditions.*

1. Participant: A Jolie *service*, say `Original`, exposing the operation subject to refactoring, say `op`, as part of an interface, say `OriginalAPI`.
2. Precondition: `op` is a retrieval operation whose response type contains an ordered collection of items to be paginated.

*Refactoring steps.*

1. Change the definition of `op` in `OriginalAPI` to:
  - (a) Extend the *request type* with metadata fields specifying the offset of the requested page, the `limit` of items per page, and the `sort-criterion`, if more than one order exists for items in the collection;
  - (b) Extend the *response type* with fields describing the response page such as the `page number offset`, `items per page limit`, `sort-criterion`, and `total number of pages`.
  - (c) Add a faulty response message `InvalidPageRequest` in case of invalid page metadata.
2. Change the implementation of `op` to:
  - (a) Validate the page metadata fields (and reply immediately with `InvalidPageRequest` in case of failure).

- (b) Paginate the requested data, possibly by leveraging features of the database query language (like `OFFSET` and `LIMIT` for SQL).
- (c) Reply with the requested page and its metadata.

*Postconditions.*

1. Calling `op` to request page with a given `offset` and `size limit` results into the items of the collection returned by the original `op` from position `offset * limit` to position `offset * limit + limit`.

*Discussion.* Delegating the pagination to the query language of the database in use achieves efficiency score  $E \star \star \star$ . However, since it also modifies the implementation of the specific operation, we obtain maintainability  $M \star \star \star$  and isolation  $I \star \star \star$ . The overall EMI score is therefore  $E \star \star \star M \star \star \star I \star \star \star$ .

*Considerations on alternative implementations.* The design smells that motivate the introduction of the `PAGINATION` pattern are about poor efficiency and thus the Ad-hoc/Internal strategy is a natural choice. If Ad-hoc/Internal is undesirable, other strategies can still be adopted at the cost of high decreases in efficiency. The key problem is distribution. Choosing an Adjacent strategy would still imply that the original API provider fetches all data from its database, but at least this would be ‘cut’ by the refactored API provider before it is sent back to clients. The same holds for an External strategy, but in this case we would pay also the cost of network communication (of the whole data set) between the refactored and original API providers.

## 6 Conclusion

We have introduced the EMI framework, the first conceptual framework for navigating the implementation aspects of API refactorings. While broad and technology-agnostic, our scores are informative when it comes to key design decisions on the implementation of API patterns.

Our study opens up at least two interesting lines of future work.

First, in line with previous work [25], we have focused on presenting API refactorings that *add* a pattern. However, our Adjacent and External strategies make it immediate to *remove* a pattern later on. We think that enabling the modular activation and deactivation of patterns is an interesting direction.

The second line of future work deals with exploring additional aspects on top of efficiency, maintainability, and isolation. These aspects are in line with a previous survey on what qualities are important in practice, but there are also others that merit consideration, like scalability and usability [3, 24]. We think that scalability would be a first natural extension of our framework, as it is closely related to efficiency and isolation but not completely captured by them.

**Acknowledgements** We thank Sandra Greiner for the useful discussions and feedback on a draft of this paper.

## References

1. Bandura, A., Kurilenko, N., Mazzara, M., Rivera, V., Safina, L., Tchitchigin, A.: Jolie community on the rise. In: Procs. SOCA. pp. 40–43. IEEE Computer Society, Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331. (2016). <https://doi.org/10.1109/SOCA.2016.16>
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Professional, 3rd edn. (2012)
3. Bogner, J., Wójcik, P., Zimmermann, O.: How do microservice api patterns impact understandability? a controlled experiment (2024), <https://arxiv.org/abs/2402.13696>
4. Chandramouli, R.: Security Strategies for Microservices-based Application Systems. National Institute of Standards and Technology (2019). <https://doi.org/https://doi.org/10.6028/NIST.SP.800-204>, available at <https://doi.org/10.6028/NIST.SP.800-204>
5. Cruz-Filipe, L., Kostopoulou, S., Montesi, F., Vistrup, J.:  $\mu$ xl: Explainable lead generation with microservices and hypothetical answers. In: Papadopoulos, G.A., Rademacher, F., Soldani, J. (eds.) Service-Oriented and Cloud Computing - 10th IFIP WG 6.12 European Conference, ESOC 2023, Larnaca, Cyprus, October 24–25, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14183, pp. 3–18. Springer (2023). [https://doi.org/10.1007/978-3-031-46235-1\\_1](https://doi.org/10.1007/978-3-031-46235-1_1), [https://doi.org/10.1007/978-3-031-46235-1\\_1](https://doi.org/10.1007/978-3-031-46235-1_1)
6. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer (2017). [https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12), [https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12)
7. Erder, M., Pureur, P.: Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World. Elsevier Science (2015), <https://books.google.dk/books?id=xxYoCgAAQBAJ>
8. Erder, M., Pureur, P., Woods, E., Safari, a.O.M.C.: Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps. Addison-Wesley Professional (2021), <https://books.google.dk/books?id=nRZwzEACAAJ>
9. Ford, N., Parsons, R., Kua, P.: Building Evolutionary Architectures: Support Constant Change. O’Reilly Media (2017), <https://books.google.dk/books?id=qYI2DwAAQBAJ>
10. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley (2012)
11. Gabbrielli, M., Giallorenzo, S., Lanese, I., Zingaro, S.P.: A language-based approach for interoperability of IoT platforms. In: Bui, T. (ed.) Procs. HICSS. pp. 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2404 Maile Way, D307, Honolulu, HI 96822 (2018)
12. Gabbrielli, M., Martini, S., Giallorenzo, S.: Programming Languages: Principles and Paradigms, Second Edition. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2023)
13. Giaretta, A., Dragoni, N., Mazzara, M.: Joining Jolie to docker - orchestration of microservices on a containers-as-a-service layer. In: Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G. (eds.) Procs. SEDA. Advances in Intelligent Systems and Computing, vol. 717, pp. 167–175. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland (2016). [https://doi.org/10.1007/978-3-319-70578-1\\_16](https://doi.org/10.1007/978-3-319-70578-1_16)



14. Guidi, C., Anedda, P., Vardanega, T.: Paassoa: An open paas architecture for service oriented applications. In: Paoli, F.D., Pimentel, E., Zavattaro, G. (eds.) Service-Oriented and Cloud Computing - First European Conference, ESOC 2012, Bertinoro, Italy, September 19-21, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7592, pp. 208–209. Springer (2012). [https://doi.org/10.1007/978-3-642-33427-6\\_16](https://doi.org/10.1007/978-3-642-33427-6_16), [https://doi.org/10.1007/978-3-642-33427-6\\_16](https://doi.org/10.1007/978-3-642-33427-6_16)
15. Guidi, C., Maschio, B.: A Jolie based platform for speeding-up the digitalization of system integration processes. In: Microservices (2019), [https://www.conf-microservices/2019/papers/Microservices\\_2019\\_paper\\_6.pdf](https://www.conf-microservices/2019/papers/Microservices_2019_paper_6.pdf)
16. Gusmanov, K., Khanda, K., Salikhov, D., Mazzara, M., Mavridis, N.: Jolie good buildings: Internet of things for smart building infrastructure supporting concurrent apps utilizing distributed microservices. CoRR abs/1611.08995 (2016)
17. Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., Stocker, M.: Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In: Sousa, T.B. (ed.) Procs. EuroPloP. ACM, 2 Penn Plaza, Suite 701 New York New York 10121-0701 (2019). <https://doi.org/10.1145/3361149.3361164>
18. Montesi, F.: Process-aware web programming with jolie. Sci. Comput. Program. 130, 69–96 (2016). <https://doi.org/10.1016/J.SCICO.2016.05.002>, <https://doi.org/10.1016/j.scico.2016.05.002>
19. Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with Jolie. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Web Services Foundations, pp. 81–107. Springer, Springer Science+Business Media New York (2014). [https://doi.org/10.1007/978-1-4614-7518-7\\_4](https://doi.org/10.1007/978-1-4614-7518-7_4)
20. Montesi, F., Weber, J.: From the decorator pattern to circuit breakers in microservices. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018. pp. 1733–1735. ACM (2018). <https://doi.org/10.1145/3167132.3167427>, <https://doi.org/10.1145/3167132.3167427>
21. Nygard, M.: Release it!: design and deploy production-ready software (2007)
22. OASIS: Web services business process execution language version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html> (2007)
23. Oram, A.: Ballerina: A Language for Network-Distributed Applications. O’Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472 (2019)
24. Stocker, M., Zimmermann, O.: From code refactoring to API refactoring: Agile service design and evolution. In: Barzen, J. (ed.) Service-Oriented Computing. pp. 174–193. Springer International Publishing, Cham (2021)
25. Stocker, M., Zimmermann, O.: API refactoring to patterns: Catalog, template and tools for remote interface evolution. In: Proceedings of the 28th European Conference on Pattern Languages of Programs. EuroPloP ’23, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3628034.3628073>
26. Stocker, M., Zimmermann, O., Zdun, U., Lübke, D., Pautasso, C.: Interface quality patterns: Communicating and improving the quality of microservices APIs. In: Procs. EuroPloP. ACM, 2 Penn Plaza, Suite 701 New York New York 10121-0701 (2018). <https://doi.org/10.1145/3282308.3282319>
27. Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C., Stocker, M.: Interface responsibility patterns: Processing resources and operation responsibilities. In: Procs. EuroPloP. ACM, 1601 Broadway, 10th Floor New York, New York 10019, USA (2020). <https://doi.org/10.1145/3424771.3424822>
28. Zimmermann, O., Pautasso, C., Lübke, D., Zdun, U., Stocker, M.: Data-oriented interface responsibility patterns: Types of information holder resources. In: Procs.

- EuroPLoP. ACM, 1601 Broadway, 10th Floor New York, New York 10019, USA (2020). <https://doi.org/10.1145/3424771.3424821>
29. Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U.: Introduction to microservice API patterns (MAP). In: Cruz-Filipe, L., Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S. (eds.) Joint Procs. Microservies. OASICS, vol. 78. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Oktavie-Allee, 66687 Wadern, Germany (2019). <https://doi.org/10.4230/OASICS.MICROSERVICES.2017-2019.4>
  30. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U.: Interface representation patterns: Crafting and consuming message-based remote APIs. In: Procs. EuroPLoP. ACM, 2 Penn Plaza, Suite 701 New York New York 10121-0701 (2017). <https://doi.org/10.1145/3147704.3147734>
  31. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C.: Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley Signature Series (Vernon), Addison-Wesley Professional (2022)