Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 155–163
DOI: 10.15439/2025F1225
ISSN 2300-5963 ACSIS, Vol. 43

Towards a Framework for Systematic API Migrations

Nikolas Jíša
0009-0005-1551-2740
Czech Technical University in Prague
Thákurova 9, 160 00 Prague 6, Czech Republic
Email: jisaniko@fit.cvut.cz

Robert Pergl
0000-0003-2980-4400
Czech Technical University in Prague
Thákurova 9, 160 00 Prague 6, Czech Republic
Email: robert.pergl@fit.cvut.cz

***Abstract*—This work presents a framework for systematic Application Programming Interface (API) migrations that provides guidance for both manual and automated migration processes. The approach starts by matching methods from the old API version with those of the new version, followed by comparative syntactic and semantic analyses to produce migration steps. As a proof of concept, we implement the most challenging parts using structured, parameterized Artificial Intelligence (AI) queries, which also serve as a basis for evaluation. Our results demonstrate the feasibility and usefulness of this approach. Future work will focus on completing the full migration process while reducing reliance on AI due to its current limitations in reasoning. We also plan to evaluate the framework on real-world APIs to assess its effectiveness and general applicability.**

I. Introduction

IN RECENT years, technological progress has accelerated rapidly. One clear example of this is the widespread integration of the Internet into everyday life, supporting activities such as online shopping, banking, social networking, and information retrieval through Artificial Intelligence (AI) queries [1]. Kurzweil [2] expands Moore's Law by applying the idea of exponential growth not only to hardware but also to software and other technological areas, with the Internet serving as one such example. The Internet consists of many interconnected systems that rely on each other. Some of these systems are client-server applications, which are often built using Application Programming Interfaces (APIs).

The problem we focus on is API evolvability, with a particular emphasis on the challenge of API migration. Lamothe et al. [3] identified several research challenges related to API evolvability. Among these, we are especially interested in the following: **(a)** Combining textual merging with syntactic and semantic approaches, **(b)** Providing a commercially viable API migration solution, **(c)** Incorporating domain-specific information into tools, **(d)** Developing more tools to assist with Web API, **(e)** Automatically identifying factors driving API changes, **(f)** Addressing API semantics and dependencies, and **(g)** Supporting the context sensitivity of API migration tools.

Although the article [3] was published in late 2021, our extended review of the API evolvability literature, presented in section III, shows that many of these challenges continue to be highly relevant.

In this paper, our research objective is: "**To create a framework for systematic API migrations that provides guidelines for performing API migrations, whether fully manual or semi-automated**." Our approach targets scenarios where an application uses a specific version of an API and needs to migrate its API calls to a newer version. A key requirement is to know the method signatures of both the old API version and the new API version, at least at the syntactic level. However, method signatures alone allow for only basic migration guidelines. If the source code of the new API version is available, static analysis can be used to generate more detailed guidance. Furthermore, if semantic documentation artifacts are available, changes at the semantic level—even if the source code remains unchanged—can be incorporated to further improve the quality of the migration guidelines.

Apart from mapping the landscape in the area of API migration research and reporting the progress of our own API migration work, the contribution of this paper lies in presenting a high-level approach to API migration. This approach consists of several components, each of which can be implemented in different ways. Nevertheless, we do not provide implementations here.

The remainder of this paper is structured as follows. Section II presents the research methodology. Related work is discussed in section III. The core of our contribution is detailed in section IV, where we explain our approach to API migration. Section V provides the evaluation of our approach, and section VI provides further discussion. Finally, section VII concludes the paper and outlines directions for future work.

II. Research Methodology

We follow the Design Science Research Methodology (DSRM), first introduced by Hevner et al. [4] and later improved in [5]. DSRM focuses on creating and evaluating new artifacts — such as models, methods, processes, or systems — that aim to solve real-world problems. The key components are: **Environment**, the real-world context including people, organizations, technologies, and problems. It sets the requirements and limits for the research. **Knowledge Base** holds existing theories, methods, frameworks, and tools that support the research. **Design Science Research** is an

iterative process of `Design Cycles` that build and test artifacts to solve problems. It connects to the **Environment** via the `Relevance Cycle` and to the **Knowledge Base** through the `Rigor Cycle`.

This paper builds on our previous work aimed at supporting API usage and migration. We initially explored various API documentation formats, such as OpenAPI Specification (OAS) [6] and API Blueprint (APIB) [7], with the goal of designing a unified format and mapping mechanisms between common specifications. So far, we have implemented a mapping from OAS to APIB [8], with plans to develop a central model and mappings to other formats. In the approach proposed in this paper, one possible input for step **M2.b** is documentation represented in this central model. We also developed an API ontological model presented in [9], which illustrates how changes made by an API provider can impact API consumers. The model includes several key observations, such as the distinction between syntactic and semantic changes, which forms the foundation of the approach presented in this paper.

In this paper, first, we do another `Relevance Cycle` by looking at updates in API documentation and investigating recent breaking changes in open-source APIs. We provide some remarks about this part in section III and section V. Next, we do the `Rigor Cycle` by reviewing new research on API evolvability using snowballing [10], as described in section III. Finally, we do the `Design Cycle` by creating a framework for systematic API migration in section IV and provide evaluation in section V.

III. Related Work

According to Lamothe et al. in [3], the key areas of focus in API evolvability at the time of their work, published at the end of 2021, were addressing breaking API changes, enhancing API usability, and reducing API misuses. In the same paper, Lamothe et al. also identify several open challenges in the field of API evolvability. In this section, we review challenges relevant for this paper and describe some additional research works discovered by snowballing [10]. The challenges are categorized with identifiers from the original article, where "EC" stands for "Existing Challenges" (which have existing publications attempting to find solutions) and "UC" for "Unresolved Challenges" (which lack existing solutions).

EC-1*) Combining textual merging with syntactic and semantic approaches:* This challenge is based on article [11] where the context is to merge two or more versions of the same software together. The article is from year 2002, nevertheless, even today, Version Control System (VCS) tools such as Git [12], are based on textual merging without employing syntactic nor semantic approaches. In a way, we can consider API migration as merge of new version of consumed API with the consumed application itself. Therefore, this can be considered as a relevant challenge for this paper as we also employ syntactic and semantic approaches.

EC-2*) Providing a commercially viable API migration solution:* At the time of writing of [3], there had already been existing API migration solutions such as tool SemDiff [13] or ApiDiff [14]. SemDiff can recommend replacements of API calls for methods which were removed in given API based on static analysis of Abstract Syntax Trees (ASTs) corresponding to code files in VCS. ApiDiff tool is based on static analysis and similarity heuristics of Java libraries hosted on Git [12] repositories in order to detect breaking changes and non-breaking changes between two versions. ApiDiff and SemDiff have not been proven commercially viable at the time of writing of [3]. One of the problems was that, as for most API Evolvability works, there was a lack in area of evaluation, because there had not been established standardized approaches and datasets to evaluate API evolvability research works which would be in wide use [3]. Both the SemDiff and ApiDiff approaches employ static analysis taking into account syntactic information but without utilizing semantic information.

In [15] Deshpande et al. tackle the problem of API migration using multi-objective evolutionary algorithms. Their approach is not restricted to cases where a single source method is always transformed into a single target method (one-to-one mappings). Instead, it is also applicable to scenarios where one or multiple source methods are mapped to multiple target methods (one-to-many and many-to-many mappings).

In [16] Ramos et al. present the MELT system, which extracts API Consumer-side transformations by analyzing pull requests on the API side. This process combines static analysis with Natural Language Processing (NLP) of pull request descriptions and comments.

Some approaches leverage program synthesis to generate transformation procedures by using examples of API call mappings between different versions. Notable examples include the APIFIX tool introduced in [17] and the ReFazer tool introduced in [18].

In [19] Beuer-Kellner et al. propose an API migration approach that utilizes a service to manage the conversion of data structures between different API versions.

In [20], Huang et al. propose an API mapping approach called MATL, which employs transfer learning to automate API mapping without requiring knowledge of the underlying source code of the API involved.

Another approach involves having developers on the API side create transformation scripts to assist API Consumers in updating API calls between versions [3]. Similarly to our goal, the mentioned techniques deal with API migration. There already exist multiple works with employ both static and semantic information in order to automatize API Migration. The idea of our approach is to employ combination of static analysis with analysis of additional semantic information.

EC-3*) Incorporating domain-specific information into tools:* This challenge is based on [11] in which Mens points out that more research is needed for the detection and resolution of structural merge conflicts that arise in the presence of restructuring transformation, and that we need more domain-specific information because it cannot be inferred from code. Our approach also relies on additional information beyond

source code.

***EC-10**) More tools to help with Web APIs:* According to Wittern in [21], Web API providers also control runtime of APIs and can do changes anytime with severe consequences to their API consumers as opposed to Library APIs. Additionally, Web APIs often lack machine-undrestandable specifications, and data are often passed over strings. Our research idea should be applicable to both Web APIs and Library APIs, therefore it should contribute to Web APIs tooling making EC-10 a relevant challenge for our research.

***EC-16**) Dealing with API semantics and dependencies:* In [22] Amman et al. describes following calls to action:

1) *We need precise definition of API usages, considering usage properties such as usage location and call multiplicities*
2) *We need a representation of such usages that captures all code details necessary to distinguish correct usages from misuses and more precise analyzes to identify usages in code*
3) *We need detectors that retrieve sufficiently many usage examples using project-external sources, such as large project sets or code-search engines*
4) *We need detectors that go beyond native assumption that a deviation from the most-frequent usage corresponds to a misuse, but consider program semantics, such as type hierarchies and implicit dependencies between objects. We hypothesize that probabilistic models might be a way to tackle this problem*
5) *We need strategies to properly match patterns and usages in the presence of violations*
6) *We need strategies to properly handle alternative patterns for the same API*
7) *Finally, we need good ranking strategies, to reduce the cost of reviewing findings.*

In the same work Amman et al. present MuBench and MuBenchPipe as foundational tools to support repeatable and replicable studies that enable systematic evaluation and analysis of alternative approaches and strategies in order to move forward with the mentioned calls to action.

Dealing with API semantics is definitely in scope of our research.

***UC-2**) Supporting the context sensitivity of API migration tools:* It is not yet clear how to best approach the context sensitivity in API migration tools. One of the proposed directions is the incorporation of domain-specific information into tools. Our research idea is related to this challenge, because we also consider incorporation of domain-specific information in form of semantic documentation in structured API documentation formats (such as OAS).

IV. Our Approach

Our proposed approach for API migration should serve as a guideline for performing API migration for developers managing consumption of an evolving API in applications - regardless whether it is performed manually or with some automated steps. The most relevant inputs are as follows:

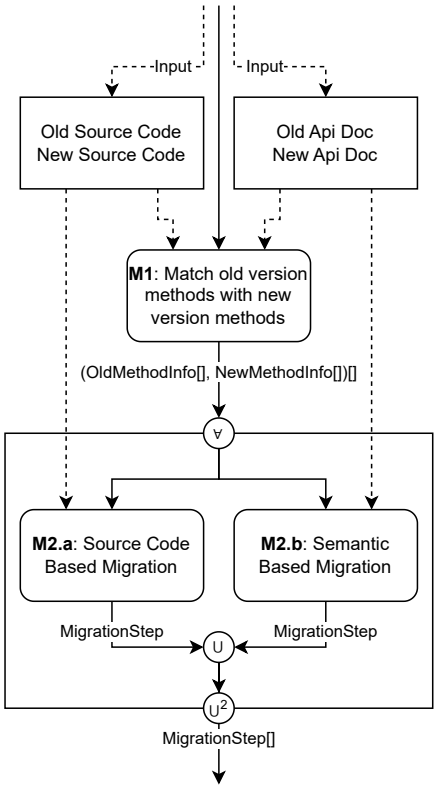

Fig. 1: Overview of our API migration approach

1) Structured API documentation for both the old and the new versions of the consumed API such as OAS documentation. **2)** Links to source code repositories (e.g., in Git), corresponding to both the old and new API versions, when available.

Upon execution, the API migration tool follows a two-step process depicted in fig. 1: **1)** Matching API methods between the old and new API versions (**M1**). **2)** Performing comparative syntax analysis (when source code is available) alongside semantic analysis for each match identified in the previous step, in order to generate a set of *MigrationStep*s (**M2.a** and **M2.b**).

MigrationStep represents a transformation applicable to old-version API method calls, specifying how to adapt them for the new version. The resulting *MigrationStep*s provide developers with a structured foundation to carry out the migration in manual or semi-automated manner.

For clarity, we present a demonstrative example comprising two versions of the same API, shown in example 4.2 and example 4.3. Both versions utilize the `Product` class defined in example 4.1, where the `[Key]` attribute — pro-

Example 4.1 (ASP.NET Core (C#)): Definition of `Product`:

```
class Product : Item
{
  public string Name { get; set; }
  public decimal Price { get; set; }
}
class Item
{
  [Key]public int Id { get; set; }
}
```

vided by Entity Framework Core [23] — designates the `Id` property as the primary key for the database. Additionally, the `dbContext` referenced in the examples corresponds to an instance of `DbContext` [24], which is used to access and interact with the database.

An example of the output from our proposed approach is shown in example 4.4, which migrates old version API calls from example 4.2 to new version API calls from example 4.3. This output is represented as JSON, where each object in the top-level array corresponds to a possible *MigrationStep*. The strings labeled "`RegexMatch`" denote regular expressions, each paired with an associated "`Replacement`" string. Within these replacement strings, "`$1`" refers to the first captured group in the match.

The primary purpose of these examples is to illustrate the concept of inputs and outputs rather than to present fully-fledged input/output artifacts.

M1: Match old version methods with new version methods

The purpose of this step is to identify correspondences between API methods in the old version and the new version of given API. Specifically, it determines which methods in the old version align with which methods in the new version, based on structural and semantic similarity. The output of this step is a collection of tuples `(OldMethodInfo[], NewMethodInfo[])[]`, where each `MethodInfo` instance encapsulates identifying and basic metadata about a given method, such as its name, return type and parameter types.

There are multiple possible strategies for implementing this matching step. We propose implementation in two parts as follows: **M1.1) Exact Signature Match**: Methods that share identical signatures — that is, matching method names, return types, and parameters — in both the old and new versions are matched directly. For example, during the migration from example 4.2 to example 4.3, only the method `DeleteProduct` is matched this way. **M1.2) Remaining Methods Match**: For methods which were not matched by the previous step, several heuristic strategies may be applied: **a)** Manual matching by the developers **b)** Matching based solely on method names **c)** Matching based on parameters and return types **d)** Matching via AST comparison, when source code is available... During the migration from example 4.2 to example 4.3, the methods `CreateProduct` and `UpdateProduct` in the old version are expected to be matched with the method `CreateOrUpdateProduct` in the new version. Similarly, the method `GetAllProducts()` from the old version should correspond to `GetProducts(decimal minPrice)` in the new version. Conversely, the method `GetProduct` from the old version does not have a matching counterpart in the new version.

Methods in the old API version that have no corresponding counterparts in the new version are classified as *deleted methods*, while those introduced only in the new version are considered *added methods*.

It is important to note that, depending on the implementation, the matching algorithm may produce false negatives—failing to correctly associate semantically equivalent methods across versions. In such cases, logically related method pairs might be misclassified as a deletion and an addition, rather than as an evolution of a single method, which could ultimately render our approach non-functional for involved methods. This underscores the necessity of maximizing the effectiveness of step **M1.2**.

Furthermore, the mapping between methods in the old and new versions is not limited to one-to-one or zero-to-one relationships. Many-to-one (M:1), one-to-many (1:N), and even many-to-many (M:N) mappings are possible, as illustrated in migration of `CreateProduct` and `UpdateProduct` in example 4.2 to `CreateOrUpdateProduct` in example 4.3. A baseline implementation that accounts for this could analyze method names by identifying segments such as "And" or "Or", and then checking whether corresponding methods exist for the parts before and after these segments. It should also verify that the parameter and return types match appropriately. A more advanced implementation could incorporate static analysis when the source code is available.

M2.a: Source Code Based Migration

The primary objective of this step is to derive the necessary migration transformations to ensure that the updated version of the API-consuming application remains syntactically correct. Specifically, it aims to determine how each method call to the old API version should be transformed into a corresponding call to the new API version, preserving equivalent application behavior. A key prerequisite for this process is access to the source code of the consumed API.

To illustrate the inputs and outputs, consider migrating method calls from `GetAllProducts()` in example 4.2 to method calls `GetProducts(decimal minPrice)` in example 4.3. This step should establish that calls to `GetAllProducts()` correspond to calls to `GetProducts(decimal.MinValue)` in the new version and generate the appropriate *MigrationStep*, as demonstrated in example 4.4.

There are several possible approaches to implementing this step. Below, we outline three representative strategies:

1) **Manual Transformation**: In this baseline approach, the developer manually inspects the old and new versions of each API method and defines the required transformations manually.

Example 4.2 (ASP.NET Core (C#)): Old version of API:

```
Product? GetProduct(int productId) =>
 ↪  dbContext.Products.FirstOrDefault(p
 ↪  => p.Id == productId);
IEnumerable<Product> GetAllProducts() =>
 ↪  dbContext.Products;
Product? CreateProduct(Product product)
 ↪  {
  dbContext.Products.Add(product);
  dbContext.SaveChanges();
  return product;
}
Product? UpdateProduct(Product product)
 ↪  {
  dbContext.Products.Update(product);
  dbContext.SaveChanges();
  return product;
}
void DeleteProduct(int productId) {
  var product =
   ↪  dbContext.Products.FirstOrDefault(p
   ↪  => p.Id == productId);
  if (product is null)
    return;
  var result =
   ↪  dbContext.Products.Remove(product);
  dbContext.SaveChanges();
}
```

Example 4.3 (ASP.NET Core (C#)): New version of API:

```
IEnumerable<Product> GetProducts(decimal
 ↪  minPrice) =>
 ↪  dbContext.Products.Where(p =>
 ↪  p.Price >= minPrice);

Product? CreateOrUpdateProduct(Product
 ↪  product) {
  if (product.Id == default)
    dbContext.Products.Add(product);
  else
    dbContext.Products.Update(product);
     ↪  ;
  dbContext.SaveChanges();
  return product;
}


void DeleteProduct(int productId) {
  var product =
   ↪  dbContext.Products.FirstOrDefault(p
   ↪  => p.Id == productId);
  if (product is null)
    return;
  var result =
   ↪  dbContext.Products.Remove(product);
  dbContext.SaveChanges();
}
```

Example 4.4 (ASP.NET Core (C#)): Output of our approach:

```
[{
  "OldMethodSignatures": ["Product? CreateProduct(Product product)", "Product?
 ↪  UpdateProduct(Product product)"],
  "NewMethodSignatures": ["Product? CreateOrUpdateProduct(Product product)"],
  "ReplacementRules": [{"RegexMatch": "CreateProduct\((\w+)\)", "Replacement":
 ↪  "CreateOrUpdateProduct($1)"},
    {"RegexMatch": "UpdateProduct\((\w+)\)", "Replacement":
    ↪  "CreateOrUpdateProduct($1)"}]
},
{
  "OldMethodSignatures": ["IEnumerable<Product> GetAllProducts()"],
  "NewMethodSignatures": ["IEnumerable<Product> GetProducts(decimal minPrice)"],
  "ReplacementRules": [{"RegexMatch": "GetAllProducts\(\)", "Replacement":
 ↪  "GetProducts(decimal.MinValue)"}]
}
{
  "OldMethodSignatures": ["Product? GetProduct(int productId)"],
  "ReplacementRules": [{"RegexMatch": "GetProduct\((\w+)\)", "Replacement":
 ↪  "GetProducts(decimal.MinValue).FirstOrDefault(p => p.Id == $1)"}]
}]
```

2) **Static analysis**: This strategy compares the ASTs of the old and new method implementations. While potentially precise, it poses significant technical challenges due to the complexity of code analysis and the wide variety of programming language constructs.
3) **AI-Based Methods**: Advances in AI, particularly large language models, offer promising opportunities for semi-automated transformation. Although current AI systems have limitations— especially in tasks requiring deep reasoning [25] — our experience with modern tools such as ChatGPT [26] indicates they can effectively address transformation problems, provided the task is well-specified.

***M2.b**: Semantic Based Migration*

The purpose of this step is to derive the migration steps necessary to ensure that the migrated version of the API-consuming application remains semantically correct. This includes preserving the intended meaning of method inputs and outputs.

To illustrate the inputs and outputs of this step, consider a scenario where the `Price` property of a `Product` is represented in US dollars in the older version, but in euros in the newer version. If this change is not properly accounted for, it can cause misinterpretation of price values, leading to errors. Importantly, such semantic differences are usually not detectable from the source code alone, even though they may sometimes be suggested by comments or naming conventions. To address this limitation, we propose extracting semantic metadata from structured documentation artifacts, such as OAS. In OAS, `Product` would be defined as a component schema, and its `Price` property could be annotated using an OAS extension (e.g., `x-semantic-meaning`), with values set to `USD` in the old version and `EUR` in the new version. Example of the `USD`-annotated OAS schema component for `Product` is shown inexample 4.5. The output of our approach would be a set of *MigrationSteps* for each method utilizing the `price` of `Product`. For instance, migrating the `GetAllProducts` method from using `USD` to `EUR` pricing would yield a *MigrationStep* as illustrated in example 4.6.

Example 4.5 (OAS JSON): Product in component schema:

```
{
  "type": "object",
  "properties": {
    "id": {"type": "string"},
    "name": {"type": "string"},
    "price": {"type": "number",
     ↪  "x-semantic-meaning": "USD"}
  }
}
```

Example 4.6 (JSON): MigrationStep for `Price` change from `USD` to `EUR`:

```
{
  "OldMethodSignatures":
   ↪  ["IEnumerable<Product>
   ↪  GetAllProducts()"],
  "NewMethodSignatures":
   ↪  ["IEnumerable<Product>
   ↪  GetAllProducts()"],
  "OldSemanticMeaning": {"Path":
   ↪  "Product.Price", "Value": "USD"},
  "NewSemanticMeaning": {"Path":
   ↪  "Product.Price", "Value": "EUR"}
}
```

A baseline implementation of this step could involve simply checking whether the semantic annotations have changed and, if so, requesting manual intervention. More advanced approaches might attempt to perform semantic migration automatically. However, such techniques are beyond the scope of this paper and represent a promising direction for future research.

V. Evaluation

A. API changes for evaluation

Initially, we aimed to identify an open-source API with a well-documented history of breaking changes to evaluate our approach. However, this task proved more difficult than anticipated. Although we identified some promising candidates, clearly documented and representative breaking changes among real-world open-source APIs were generally scarce. One example of a promising API for evaluation purposes is Elasticsearch [27], which documents several breaking changes - such as the one described and implemented in [28] where our approach would likely be effective. Nevertheless, the complexity of many APIs hindered analysis, especially when changes were undocumented or lacked sufficient context. Therefore, we decided to create our own API changes based on examples 4.2 and 4.3, which are listed in table I.

As our approach is not yet fully implemented, we conducted an evaluation focusing on the implementability and applicability of each individual step. In some cases, this evaluation was supported by leveraging AI tools, specifically ChatGPT [26]. The results of this assessment are summarized in table II.

```
I am consuming old version of API in my
 ↪  application and I have to migrate it to
 ↪  use the new version. The API is written
 ↪  in `{{API_FRAMEWORK_OR_LANGUAGE}}`. I
 ↪  need to know which steps to take in my
 ↪  API-consuming application created in
 ↪  `{{API_CONSUMER_FRAMEWORK_OR_LANGUAGE}}`,
 ↪  specifically, what should I replace the
 ↪  calls to `{{OLD_METHOD_SIGNATURES}}` from
 ↪  old version to migrate it to calls to new
 ↪  `{{NEW_METHOD_SIGNATURES}}` with behavior
 ↪  in terms of input-output mapping being
 ↪  intact? Mark the source code I should use
 ↪  in place of calls to
 ↪  `{{OLD_METHOD_SIGNATURES}}` - i.e. exact
 ↪  string I should use as replacement for
 ↪  `{{OLD_METHOD_CALLS}}` - in the answer
 ↪  between XML tags `<REPLACEMENT>` and
 ↪  `</REPLACEMENT>`.

Definition of relevant structures:
`{{SHARED_DATA_TYPE_DEFINITIONS}}`

Old version source code:
`{{OLD_CODE}}`

New version source code:
`{{NEW_CODE}}`
```

Fig. 2: AI query implementation of **M2.a** step

B. Test implementation of *M2.a* over AI

Our implementation of **M2.a** for AI-based processing is defined by a structured AI query that accepts the following self-explanatory parameters:

1) `API_FRAMEWORK_OR_LANGUAGE`
2) `API_CONSUMER_FRAMEWORK_OR_LANGUAGE`
3) `OLD_METHOD_SIGNATURES`
4) `NEW_METHOD_SIGNATURES`
5) `OLD_METHOD_CALLS`
6) `SHARED_DATA_TYPE_DEFINITIONS`
7) `OLD_CODE`
8) `NEW_CODE`

The structure of this query is illustrated in fig. 2, where parameters are enclosed using double curly braces.

We used the examples of API method changes from table I to create AI queries, which we submitted to ChatGPT using the ChatGPT-4 Turbo engine [26]. The `REPLACEMENT` parts of the generated responses are listed in table III. The responses were generally satisfactory. However, certain changes may require additional manual adjustments. In general, developers may not be aware of which modifications necessitate further intervention, and thus must manually review all changes. This limitation aligns with the intended scope of our approach, which is designed to assist—rather than fully automate—manual API migration at this stage. Nevertheless, minimizing manual effort remains a key objective, representing an important direction for future research. Also, for cases involving the weakening of preconditions or postconditions, the suggested

TABLE I: API changes for evaluation

Identifier	Change Kind	Version A	Version B
C1	Create / Remove Method **C1**		`IEnumerable<Product> GetProductsWithPriceLowerThan(decimal price) => dbContext.Products.Where(p => p.Price < price);`
C2	Add / Remove Parameter	`IEnumerable<Product> GetProducts() => dbContext.Products`	`IEnumerable<Product> GetProducts(decimal minPrice) => dbContext.Products.Where(p => p.Price >= minPrice)`
C3	Change Parameter Type to More / Less Abstract	`Product? UpdateProduct(Product product) { dbContext.Products.Update(product); dbContext.SaveChanges(); return product; }`	`Product? UpdateProduct(Item item) { dbContext.Items.Update(item); dbContext.SaveChanges(); return item; }`
C4	Data Coupling to / from Stamp Coupling	`Product? UpdateProduct(Product product) { dbContext.Products.Update(product); dbContext.SaveChanges(); return product; }`	`Product? UpdateProduct(int id, string name, decimal price) { var dbProduct = dbContext.Products.First(p => p.Id == id); dbProduct.Name = name; dbProduct.Price = price; dbContext.SaveChanges(); return dbProduct; }`
C5	Stronger / Weaker Pre-Condition with Exception	`Product? UpdateProduct(Product product) { dbContext.Products.Update(product); dbContext.SaveChanges(); return product; }`	`Product? UpdateProduct(Product product) { if (product.Price < 0) throw new ArgumentException(); dbContext.Products.Update(product); dbContext.SaveChanges(); return product; }`
C6	Stronger / Weaker Pre-Condition with Null	`Product? UpdateProduct(Product product) { dbContext.Products.Update(product); dbContext.SaveChanges(); return product; }`	`Product? UpdateProduct(Product product) { if (product.Price < 0) return null; dbContext.Products.Update(product); dbContext.SaveChanges(); return product; }`
C7	Stronger / Weaker Post-Condition with Exception	`Product? GetProduct(int productId) => dbContext.Products.First(p => p.Id == productId);`	`Product? GetProduct(int productId) { var result = dbContext.Products.First(p => p.Id == productId); if (result.Price < 0) throw new InvalidOperationException(); return result; }`
C8	Stronger / Weaker Post-Condition with Null	`Product? GetProduct(int productId) => dbContext.Products.First(p => p.Id == productId);`	`Product? GetProduct(int productId) { var result = dbContext.Products.First(p => p.Id == productId); if (result.Price < 0) return null; return result; }`
C9	Method composition / decomposition	`Product? CreateProduct(Product product) { dbContext.Products.Add(product); dbContext.SaveChanges(); return product; } Product? UpdateProduct(Product product) { dbContext.Products.Update(product); dbContext.SaveChanges(); return product; }`	`Product? CreateOrUpdateProduct(Product product) { if (product.Id == default) dbContext.Products.Add(product); else dbContext.Products.Update(product);; dbContext.SaveChanges(); return product; }`
C10	Change Primitive Type from / to Smaller	`Product? GetProduct(int productId) => dbContext.Products.FirstOrDefault(p => p.Id == productId)`	`Product? GetProduct(long productId) => dbContext.Products.FirstOrDefault(p => p.Id == productId)`
C11	Add / Remove Field to / from Parameter	`class Filter { decimal MinPrice { get; set; } decimal MaxPrice { get; set; } } IEnumerable<Product> GetProducts() => dbContext.Products; IEnumerable<Product> GetProducts(Filter filter) { var allProducts = GetProducts(); var result = allProducts.Where(p => p.Price >= filter.MinPrice && p.Price <= filter.MaxPrice); return result; }`	`class Filter { string SearchPhrase { get; set; } decimal MinPrice { get; set; } decimal MaxPrice { get; set; } } IEnumerable<Product> GetProducts() => dbContext.Products; IEnumerable<Product> GetProducts(Filter filter) { var allProducts = GetProducts(); var result = allProducts.Where(p => p.Price >= filter.MinPrice && p.Price <= filter.MaxPrice && p.Name.Contains(filter.SearchPhrase)); return result; }`

TABLE II: Evaluation summary

Step	Comment
M1.1	Matching methods of identical signatures is straightforward and thus excluded from the evaluation.
M1.2	We have confirmed that ChatGPT can successfully perform this step using the example presented in section IV.
M2.a	This step is regarded as the most challenging, and consequently, we implemented it using a structured ChatGPT query described in section V-B, which produced satisfactory results.
M2.b	We have confirmed that ChatGPT can successfully execute this step using the example outlined in section IV.

changes could be undesirable. One possible improvement would be to explicitly request `RegexStrings` with corresponding `Replacement` values, so that they could be directly used in the resulting *MigrationSteps*, as demonstrated in example 4.4.

VI. Discussion

We fulfill the research objective outlined in section I by developing a framework for systematic API migration (manual or semi-automated), which is described in section IV and later evaluated in section V. However, several aspects of our proposed approach require further discussion.

Firstly, it would be valuable to compare our API migration methodology with existing approaches. However, as noted by Lamothe et al. [3], comparing API migration techniques remains an open and unresolved challenge. Additionally, we currently lack a complete implementation of our methodology. Instead, we have implemented and evaluated only individual components using AI queries. It would be interesting to see what a full implementation would look like and how it would perform on real-world APIs.

Secondly, during the evaluation, we repeatedly observed that changes preserving backward compatibility might not always be desirable. This is particularly true for modifications that weaken preconditions or postconditions. These observations prompt an important question: how should we determine

TABLE III: Summary of AI responses

Change Identifier	Direction	Response part between "<REPLACEMENT>" and "</REPLACEMENT>"	Manual steps needed?	Comment
C1	A to B	N/A	No	No changes necessary
C1	B to A	N/A	Yes	Injecting code of the removed method might work
C2	A to B	`GetProducts(0)`	No	`decimal.MinValue` would be better than `0`
C2	B to A	`GetProducts().Where(p => p.Price >= minPrice)`	Yes	There needs to be `minPrice` variable
C3	A to B	`(Product?)UpdateProduct(product)`	No	
C3	B to A	`UpdateProduct((Product)item)`	No	
C4	A to B	`UpdateProduct(product.Id, product.Name, product.Price)`	No	
C4	B to A	`new Product { Id = id, Name = name, Price = price }`	No	
C5	A to B	Same as before	Yes	Exception handling might be desired

C5	B to A	`if (product.Price < 0)`
		`  throw new ArgumentException();`
		`UpdateProduct(product)`
			Yes	Keeping the same behavior as before might be undesired here

C6	A to B	Same as before	Yes	Null handling might be desired
C6	B to A	`product.Price < 0 ? null : UpdateProduct(product)`	Yes	Keeping the same behavior as before might be undesired here

C7	A to B	`try {`
		`  GetProduct(productId);`
		`} catch (InvalidOperationException) {`
		`  return null;`
		`}`
			Yes	Should have proper exception handling

C7	B to A	`var product = GetProduct(productId);`
		`if (product.Price < 0)`
		`  throw new InvalidOperationException();`
			Yes	Keeping the same behavior as before might be undesired here

C8	A to B	`var product = GetProduct(productId);`
		`if (product == null)`
		`  throw new InvalidOperationException("Product price is invalid");`
			Yes	Throwing an exception might be undesired

C8	B to A	`var product = GetProduct(productId);`
		`if (product.Price < 0)`
		`  return null;`
			Yes	Keeping the same behavior as before might be undesired here

C9	A to B	`CreateOrUpdateProduct(product);`	No	
C9	B to A	`(product.Id == default ? CreateProduct(product) : UpdateProduct(product));`	No	
C10	A to B	`GetProduct((long)productId)`	No	
C10	B to A	`GetProduct((int)productId);`	Yes	There might be some data loss like this
C11	A to B	`GetProducts(new Filter { MinPrice = filter.MinPrice, MaxPrice =` `↪  filter.MaxPrice }).Where(p => p.Name.Contains(filter.SearchPhrase))`	No	
C11	B to A	`GetProducts(new Filter { MinPrice = filter.MinPrice, MaxPrice =` `↪  filter.MaxPrice, SearchPhrase = "" })`	No	

which changes are desirable and which are not? Moreover, is it possible to evaluate the desirability of such changes in an automated or semi-automated manner? These questions remain open for future research.

Thirdly, several steps in our approach are defined at a high level of abstraction, and it remains uncertain whether all of them can be feasibly implemented. For instance, step **M2.a** is intended to leverage static analysis, potentially supported by AI tools. However, the precise mechanism and its practical efficiency are still unclear. Furthermore, methods involving AI introduce additional challenges related to effectiveness and reproducibility, due to their inherently black-box nature.

Next, although our approach draws on many ideas from the research reviewed in section III, we have done little work on integrating it with existing solutions. This remains an interesting direction for future research.

Finally, since we successfully demonstrated the applicability of AI in step **M2.a**, a natural question arises—could an AI-based approach be employed for the entire API migration process, leveraging source code, structured API documentation, and other related artifacts? While we have not yet explored this possibility in detail, we hypothesize that such an approach could be feasible to some extent. Nonetheless, considering the limitations of current AI systems, particularly their challenges with consistent and transparent reasoning, we propose a more cautious direction. In future work, we aim to reduce reliance on AI and instead focus on more deterministic and interpretable alternatives.

VII. Conclusion

In this work, we presented a systematic framework for API migrations. The goal of this framework is to guide developers during migration, whether they choose to do it manually or with the help of automated tools. The migration process starts by matching methods from the old version of the API with those in the new version. It then continues with both comparative syntactic and semantic analyses to produce clear and structured migration steps.

As part of our evaluation, we implemented the comparative syntactic analysis as a structured AI query. We observed that this step was the most technically challenging part of the process. Nevertheless, our working prototype shows that it is not only feasible but also valuable in practice. It lays a solid foundation for future improvements of the framework.

In future work, we plan to complete the full implementation of the migration process. One of our aims is to reduce reliance on AI, since current models still struggle with consistent and reliable reasoning across different migration scenarios.

Another important direction for future work is to apply our framework to real-world APIs, rather than relying only on synthetic examples. This will allow us to evaluate how well the framework performs in practical environments and to what extent it can be generalized to different kinds of APIs.

Statement on the use of AI

AI technologies (ChatGPT [26]) were used to improve the language of the paper.

Acknowledgements

This research was supported by the grant of Czech Technical University in Prague No. SGS23/206/OHK3/3T/18.

References

[1] L. Rainie and B. Wellman, "The Internet in Daily Life: The Turn to Networked Individualism," in *Society and the Internet*. Oxford University Press, Jul. 2019, pp. 27–42. ISBN 978-0-19-884349-8 978-0-19-187932-6. [Online]. Available: https://academic.oup.com/book/35088/chapter/299127482

[2] R. Kurzweil, "The Law of Accelerating Returns," in *Alan Turing: Life and Legacy of a Great Thinker*, C. Teuscher, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 381–416. ISBN 978-3-642-05744-1 978-3-662-05642-4. [Online]. Available: http://link.springer.com/10.1007/978-3-662-05642-4_16

[3] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A Systematic Review of API Evolution Literature," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–36, Nov. 2022. doi: 10.1145/3470133. [Online]. Available: https://dl.acm.org/doi/10.1145/3470133

[4] Hevner, March, Park, and Ram, "Design Science in Information Systems Research," *MIS Quarterly*, vol. 28, no. 1, p. 75, 2004. doi: 10.2307/25148625. [Online]. Available: https://www.jstor.org/stable/10.2307/25148625

[5] A. Hevner, "A Three Cycle View of Design Science Research," *Scandinavian Journal of Information Systems*, vol. 19, Jan. 2007.

[6] "OpenAPI Specification v3.1.1," accessed: 2025-06-26. [Online]. Available: https://spec.openapis.org/oas/latest.html

[7] "API Blueprint Specification | API Blueprint," accessed: 2025-06-26. [Online]. Available: https://apiblueprint.org/documentation/specification.html

[8] R. Pergl and N. Jísa, "Semantic Analysis of API Blueprint and OpenAPI Specification," in *Czech Technical University Prague*, A. Rocha, H. Adeli, G. Dzemyda, F. Moreira, and A. Poniszewska-Maranda, Eds., vol. 989, 2024. doi: 10.1007/978-3-031-60227-6_15. ISBN 2367-3370 pp. 172–181.

[9] N. Jíša and R. Pergl, "Towards Evolvable APIs through Ontological Analysis," in *Annals of Computer Science and Information Systems*, vol. 41. PTI, Nov. 2024. doi: 10.15439/2024f3164. ISSN 2300-5963 pp. 61–68.

[10] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. Ease '14. London, England, United Kingdom and New York, NY, USA: Association for Computing Machinery, 2014. doi: 10.1145/2601248.2601268. ISBN 978-1-4503-2476-2

[11] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002. doi: 10.1109/TSE.2002.1000449. [Online]. Available: http://ieeexplore.ieee.org/document/1000449/

[12] "Git," accessed: 2025-06-28. [Online]. Available: https://git-scm.com/

[13] B. Dagenais and M. P. Robillard, "SemDiff: Analysis and recommendation support for API evolution," in *2009 IEEE 31st International Conference on Software Engineering*. Vancouver, BC, Canada: IEEE, 2009. doi: 10.1109/ICSE.2009.5070565. ISBN 978-1-4244-3453-4 pp. 599–602. [Online]. Available: http://ieeexplore.ieee.org/document/5070565/

[14] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "APIDiff: Detecting API breaking changes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso: IEEE, Mar. 2018. doi: 10.1109/SANER.2018.8330249. ISBN 978-1-5386-4969-5 pp. 507–511. [Online]. Available: http://ieeexplore.ieee.org/document/8330249/

[15] N. Deshpande, M. W. Mkaouer, A. Ouni, and N. Sharma, "Third-party software library migration at the method-level using multi-objective evolutionary search," *Swarm and Evolutionary Computation*, vol. 84, p. 101444, Feb. 2024. doi: 10.1016/j.swevo.2023.101444. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S221065022300216X

[16] D. Ramos, H. Mitchell, I. Lynce, V. Manquinho, R. Martins, and C. L. Goues, "MELT: Mining Effective Lightweight Transformations from Pull Requests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Luxembourg, Luxembourg: IEEE, Sep. 2023. doi: 10.1109/ASE56229.2023.00117. ISBN 979-8-3503-2996-4 pp. 1516–1528. [Online]. Available: https://ieeexplore.ieee.org/document/10298355/

[17] X. Gao, A. Radhakrishna, G. Soares, R. Shariffdeen, S. Gulwani, and A. Roychoudhury, "APIfix: output-oriented program synthesis for combating breaking changes in libraries," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021, publisher: ACM New York, NY, USA.

[18] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning Syntactic Program Transformations from Examples," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Buenos Aires: IEEE, May 2017. doi: 10.1109/ICSE.2017.44. ISBN 978-1-5386-3868-2 pp. 404–415. [Online]. Available: http://ieeexplore.ieee.org/document/7985680/

[19] L. Beurer-Kellner, J. Von Pilgrim, C. Tsigkanos, and T. Kehrer, "A Transformational Approach to Managing Data Model Evolution of Web Services," *IEEE Transactions on Services Computing*, pp. 1–1, 2022. doi: 10.1109/TSC.2022.3144613. [Online]. Available: https://ieeexplore.ieee.org/document/9689952/

[20] Z. Huang, J. Chen, J. Jiang, Y. Liang, H. You, and F. Li, "Mapping APIs in Dynamic-typed Programs by Leveraging Transfer Learning," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–29, May 2024. doi: 10.1145/3641848. [Online]. Available: https://dl.acm.org/doi/10.1145/3641848

[21] E. Wittern, "Web APIs - challenges, design points, and research opportunities: invited talk at the 2nd international workshop on API usage and evolution (WAPI '18)," in *Proceedings of the 2nd International Workshop on API Usage and Evolution*. Gothenburg Sweden: ACM, Jun. 2018. doi: 10.1145/3194793.3194801. ISBN 978-1-4503-5754-8 pp. 18–18. [Online]. Available: https://dl.acm.org/doi/10.1145/3194793.3194801

[22] S. Amann, H. Nguyen, S. Nadi, T. Nguyen, and M. Mezini, "A Systematic Evaluation of Static API-Misuse Detectors," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 45, no. 12, pp. 1170–1188, Dec. 2019. doi: 10.1109/TSE.2018.2827384

[23] "Overview of Entity Framework Core - EF Core," accessed: 2025-07-11. [Online]. Available: https://learn.microsoft.com/en-us/ef/core/

[24] "DbContext Class (Microsoft.EntityFrameworkCore)," accessed: 2025-07-11. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext?view=efcore-9.0

[25] G. Gendron, Q. Bao, M. Witbrock, and G. Dobbie, "Large Language Models Are Not Strong Abstract Reasoners," in *IJCAI Int. Joint Conf. Artif. Intell.*, Larson K., Ed. International Joint Conferences on Artificial Intelligence, 2024. ISBN 10450823 (ISSN); 978-195679204-1 (ISBN) pp. 6270–6278, journal Abbreviation: IJCAI Int. Joint Conf. Artif. Intell. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85204293915&partnerID=40&md5=b5ebdcbbf62f13f3c21b695461a5ab2d

[26] "ChatGPT," accessed: 2025-07-11. [Online]. Available: https://chatgpt.com

[27] "elastic/elasticsearch," Jul. 2025, accessed: 2025-07-11. [Online]. Available: https://github.com/elastic/elasticsearch

[28] elastic, "Store outcome values in servicemetrics 'transaction.success_count' by carsonip · Pull Request #9791 · elastic/apm-server," accessed: 2025-07-11. [Online]. Available: https://github.com/elastic/apm-server/pull/9791