Towards Fully Synthetic Schema-based Tabular Test Data Generation in Estonian E-Government Settings

Maj-Annika Tammisto[1][0009−0000−6516−1102], Dietmar Pfahl[1][0000−0003−2400−501X], Faiz Ali Shah[1][0000−0003−1233−5425], and Daniel Rodriguez[2][0000−0002−2887−0185]

[1] Institute of Computer Science, University of Tartu, Tartu, Estonia
[2] Department of Computer Science, University of Alcala, Madrid, Spain

Abstract. Testing of e-government services relies on privacy-preserving synthetic test data that are as similar as possible to actual real-life raw data, but also satisfy the requirements of the test cases. Obtaining such test data in Estonian e-government settings is a resource-intensive and largely manual process. This work addresses the challenges of the current process and suggests a novel synthetic test data generation approach that is largely automated, does not require access to real-life raw data, and generates realistic synthetic test data for Estonian e-government settings. We validate the Proof of Concept of our novel synthetic test data generation approach in real-life-like settings. We conclude that the approach can already generate synthetic data for simpler test cases, but needs additional work to cover more complex test cases that require synthetic test data to be compatible across different data services.

Keywords: Synthetic Test Data · Data Synthesis · Synthetic Data Generation · Rule-based Generation · Schema-based Generation · Tabular Data.

1 Introduction

Digital government services, also referred to as e-government services, have been a conventional part of Estonian citizens' and residents' lives since the early 2000s [1]. For Estonian citizens and residents, it is taken for granted that they can submit pre-filled annual tax reports online with just a few clicks, drive without carrying a physical driving license, and even complete divorce proceedings over the internet without needing to be physically present with their spouse.

The e-government services rely on secure data exchange that seamlessly runs in the background and aggregates information from various government institutions. This enables, for example, the automatic inclusion of relevant data in tax reports, real-time access to driver information for law enforcement, and the efficient processing of online divorce proceedings by family officials. The framework used for secure data exchange in Estonian e-government settings uses a local

implementation of the X-Road®[3] technology, called the X-tee[4]. Each member of this framework can act as a data service client requesting data, a data service provider providing data, or both. Data services that are used for requesting and providing data over the X-tee data exchange framework are, in nature, web services that use Simple Object Access Protocol (SOAP) and Representational State Transfer Protocol (REST) interaction style. The scope of data that a data service client can request from a data service provider is limited by the structure and description of data services that the data service provider provides.

Every e-government service is developed and tested before it is made available for citizens and residents. An e-government service that relies on input data received from data services when made available to citizens and residents also requires input data while it is tested, with the exception that actual, real-life raw data can´t be accessed and used for testing for data privacy reasons. Therefore, there is a constant need for realistic synthetic test data that is consistent between different data services. The synthetic test data often needs to be created ad hoc so that development teams can meet their deadlines. As we have demonstrated in [2], generating synthetic test data upon request is currently a largely manual and lengthy process. The challenges that users of the current synthetic test data generation process face can be mitigated by partly automating the process and, therefore, reducing the time and resources needed for ad hoc synthetic test data generation.

In [3], we identified and classified existing approaches that can be used to generate synthetic test data without direct access to actual real-life raw data. Existing approaches that can be used to generate complex tabular synthetic data without requiring real-life raw data as input can be classified as either specification-based or rule-based synthetic data generation approaches. However, formal specifications are usually not available, and the limitation of rule-based synthetic data generation approaches preventing their wider implementation in e-government settings is that they rely on extensive manual work and substantial domain knowledge in the rule definition phase. This makes existing rule-based synthetic data generation approaches inefficient and expensive to use in e-government settings. The solution is, therefore, to develop and validate a novel rule-based synthetic data generation approach where not only the synthetic data generation itself, but also the prior definition of rules (and constraints, if needed) requires as little manual work from humans as possible.

Our contribution includes the Proof of Concept (PoC) of a novel rule-based synthetic data generation approach for data services that use the SOAP interaction style. We use Large Language Models (LLMs) to derive rules and constraints from data service descriptions and relevant documents, and to generate synthetic test data for the respective data services with minimal intervention by humans. We validate the PoC by generating synthetic test data for a playground where an application uses input from data services that are provided in the Estonian e-government settings by different government entities.

[3] https://x-road.global/
[4] https://x-tee.ee/en/home

The rest of this paper is structured as follows: Section 2 gives an overview of previous work related to ours. Section 3 describes the steps and activities of our novel synthetic test data generation approach. Section 4 explains the validation process and defines our Research Question (RQ) that guides the validation. Section 5 describes the validation results and answers the RQ. Section 6 discusses the results, explains the limitations of the PoC of our approach, and provides insight into future work. Finally, Section 7 concludes this paper.

2 Related work

Creating synthetic test data automatically from code or descriptions is not a new idea. Automating unit test generation by exploring the input space and producing effective test cases for given programs has been studied for decades [4], [5]. Approaches for test case generation from WSDL definitions [6], OpenAPI specifications [7], or both [8] have also been suggested for many years. Still, the complexity and variety of data generated with these methods have always been too limited for high-level test cases. The vast development of LLMs has only recently allowed us to reuse the idea and, for the first time, to create complex, versatile, and realistic synthetic test data without accessing the real-life raw data and using it as training data.

The potential of using LLMs to extract business rules that are defined in natural language and to use them for test case and test data generation has also not gone unnoticed. Recent contributions are aiming to automate test case generation from business rules and requirements that are defined in natural language [9], [10]. Nevertheless, we have not been able to identify an existing approach that combines both a web service specification and unstructured business rules as input, considers the need to generate consistent synthetic data for more than one web service, and does not need to access real-life raw data.

3 Approach

In Figure 1, we show a high-level overview of our PoC. In the first step, we generate the schema that contains the structure, rules, and constraints for synthetic data generation for a specific data service, as discussed in Section 3.1. The second step is to generate the synthetic data based on the previously generated schema, by doing the activities described in Section 3.2.

3.1 Schema generation

This Subsection describes the necessary inputs for the schema generation step of our PoC, as well as the activities and outputs of each activity of the *schema generation* step.

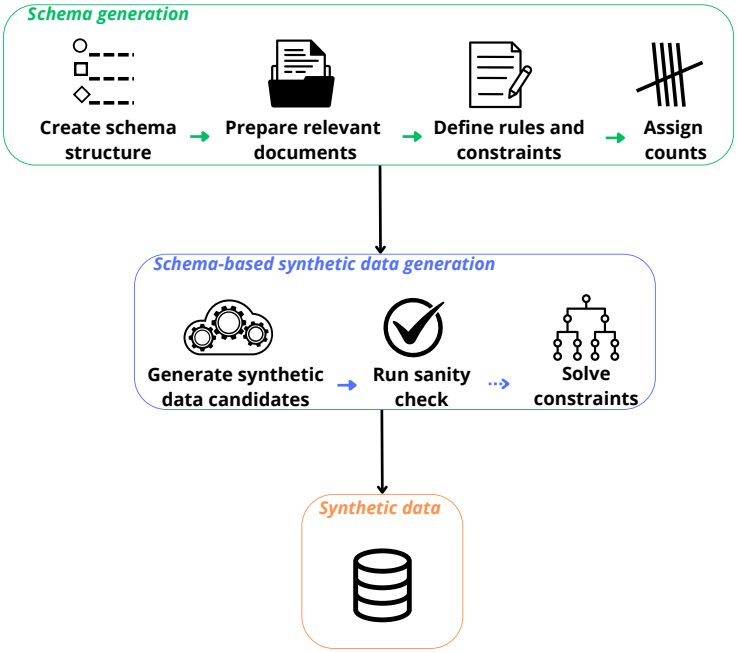

Fig. 1. Overview of the PoC

Create schema structure The *Create schema structure* activity is currently designed for data services that use the SOAP interaction style, as this interaction style is currently mostly used in Estonian e-government settings. The activity has two input variables where the values need to be specified by the user:

- WSDL description location: the location of the WSDL description that contains the description of the operation (data service) of interest. All WSDL descriptions for data services that are used in the Estonian e-government settings are publicly available in the Catalogue of all X-tee subsystems with methods and WSDL descriptions[5]. The value of the WSDL description location variable can therefore be either the location of the WSDL in the Catalogue or the location of a locally stored WSDL description file.
- Operation name: the name of our data service of interest that would, in the previous manual synthetic data generation process, be forwarded to the data service provider as reference.

Defining the two input values is the only manual contribution that is required from the user of the synthetic data generation approach. After adding the input, the schema structure will be automatically created by a Python program.

[5] https://x-tee.ee/catalogue/EE

The Python program uses the Python Zeep SOAP Client[6] to parse the WSDL description and extract all elements that belong to the defined operation, together with the required data types of their values in the data service. A JSON schema is composed of the extracted elements and data types, following the sequence of elements in the data service. The element names are listed as property names with the extracted data type specified as the type of each respective property in the JSON schema. The JSON schema, which contains the data service structure, element names of the data service as property names, and allowed data types as property types, is hereinafter referred to as the schema structure and as the output of the *Create Schema Structure* activity.

The *Create Schema Structure* activity can be summarized as follows:

```
#Create schema structure
SET wsdl_url to {WSDL location}
SET operation_name to {name of the data service}
    FOR each element in the WSDL
        IF element belongs to the data service
            ADD element name in the schema structure as
            property name
            ADD element data type in the schema structure
            as type
        ELSE
            SKIP
        END IF
    END FOR
RETURN schema structure
```

Prepare relevant documents In Estonia, all government institutions that act as data service providers must describe their information systems as well as the data that they gather and process in the catalogue of interoperability resources (RIHA)[7]. The RIHA catalogue, therefore, contains unstructured data that includes descriptions of systems, components, services, data models, semantic assets, and more. The RIHA catalogue provides a REST API interface that can be used to query the data related to a specific data service provider. In our context, the data related to the data service provider who provides our data service of interest are the relevant documents.

The *Prepare relevant documents* activity requires the name of the data service provider to be entered manually as input. The remainder of the activity is done automatically: the REST API interface provided by the RIHA catalogue is queried for all relevant data. All documents of different types (.txt, .pdf, .csv, etc.) are downloaded, converted to text, and saved as .txt files in UTF-8 encoding.

[6] https://docs.python-zeep.org/en/master/
[7] https://www.riha.ee/Avaleht

The .txt files are then split into chunks. We use the RecursiveCharacterTextSplitter by LangChain[8] with chunk size 200 and no overlap to split the documents, as this chunking method has been proven to perform well for later retrieval of information from the chunks [11]. We then create a collection for the data service of interest in a vector database. Since we use only publicly available documents, we have currently chosen the Chroma Cloud[9] vector database to reduce the technical overhead. The cloud-based database can nevertheless be easily replaced with a local vector database.

To create embeddings, or in other words, the vector representations, from each chunk, we use OpenAI's[10] text-embedding-3-large embedding model. We then add the embeddings, chunk text, and metadata (source and name of each document) to the created collection in the vector database. The output of the *Prepare relevant documents* is the collection in the vector database together with embeddings, chunk text, and metadata.

The *Prepare relevant documents* activity can be summarized as follows:

```
#Prepare relevant documents
SET document_endpoint to {document catalogue API base URL}
+{name of the data service provider}
GET all document URLs from document_endpoint
    FOR each document URL
        IF URL starts with "file://"
            EXTRACT document_id from URL
            JOIN {API base URL}+"files"+document_id
           SAVE document
           EXTRACT text from document
           SAVE text as .txt
       ELSE
           SAVE document
           EXTRACT text from document
           SAVE text as .txt
       END IF
    END FOR
```

Define rules and constraints The *Define rules and constraints* activity is a chain of five primitive LLM operations (LLM POs), as illustrated in Figure 2. Each LLM PO is designed to perform a specific task of a certain type and with default data and prompting structure [12].

PO1 takes the output of the *Create schema structure* activity as input. Each subsequent PO takes the output of the previous PO as input. In addition to that, factual queries PO2, PO3, and PO4 use the output of the *Prepare relevant documents* activity to find the k nearest neighbours to the data service name,

[8] https://www.langchain.com/
[9] https://www.trychroma.com/cloud
[10] https://openai.com/

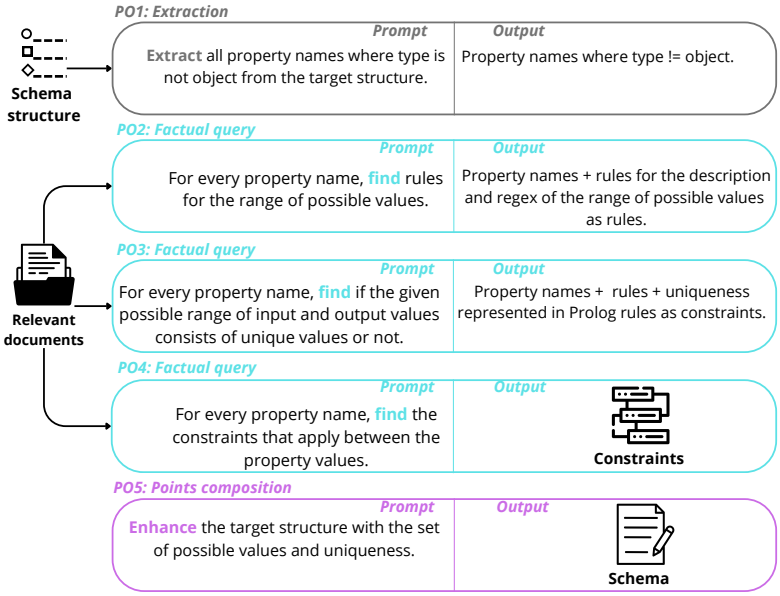

Fig. 2. The chain of LLM POs

as well as every property name in the schema structure, to provide context and improve the LLM output with Retrieval-Augmented Generation (RAG) [13].

We use the gpt-4.1 LLM from OpenAI, which is a non-reasoning model, in all LLM POs. While PO1 and PO5 are simple extraction and points composition tasks, PO2, PO3, and PO4 can be viewed as operations that also require some reasoning. Especially if the answer can´t be explicitly found in the context or in the data that has been used to teach the model. With this consideration, we have also evaluated the family or OpenAI´s gpt-5 reasoning models for PO2, PO3, and PO4. We did not notice a significant advantage in using a slower and more expensive reasoning model for PO2, PO3 and PO4. Non-reasoning and reasoning models both show inconsistencies in the rules and constraints for properties where they can´t be explicitly derived from the context. Surprisingly, we did see a 15 percent increase in inconsistencies in rules and constraints when running PO2, PO3, and PO4 repeatedly for the same data service and with the same context with the reasoning model, compared to running PO2, PO3, and PO4 repeatedly for the same data service and with the same context with the non-reasoning model. Based on these results, we concluded that the gpt-4.1 non-reasoning model is currently more suitable for our LLM POs than a reasoning model.

The first output of the *Define rules and constraints* activity is the schema structure that is enhanced with rules for the value of every property in the schema. The second output of this activity is a separate file with constraints.

The *Define rules and constraints* activity can be summarized as follows:

```
#Define rules and constraints
SET path to {name of the schema structure file}
    FOR each property in schema structure
        IF property type is object
            EXTRACT property name
        ELSE
            SKIP
        END IF
    END FOR
RETURN list of property names
SET path to {name of the directory where .txt files
are stored}
    FOR each .txt file
        SPLIT .txt file to chunks(chunk_size=x,
        chunk_overlap=y)
        CREATE embeddings(number_of_dimensions=z)
        ADD chunks and embeddings to vector database
    END FOR
QUERY vector database for {name of the data service provider}+
{name of the data service}+
every property name in the list of property names
RETURN results
FIND the set of possible values
    FOR the list of property names+results+
    {name of the data service provider}+
    {name of the data service} as context
        PROMPT LLM
    END FOR
RETURN the set of possible values
FIND the uniqueness of values
    FOR the set of possible values+results+
    {name of the data service provider}+
    {name of the data service} as context
        PROMPT LLM
    END FOR
RETURN the set of possible values + uniqueness as rules
FIND constraints
    FOR rules+results+{name of the data service provider}+
    {name of the data service} as context
        PROMPT LLM
    END FOR
```

```
RETURN constraints
ENHANCE schema structure file
    FOR each property in schema structure
        IF property type is object
            ADD rules as description of the property
        ELSE
            SKIP
        END IF
    END FOR
RETURN properties with rules
```

Assign counts From the output of the *Define rules and constraints* activity, synthetic test data can be generated by randomly specifying the size of the population only. This allows for generating a required number of sets of synthetic values from the schema where all values are within the allowed value ranges of each property, and then solving the constraints between property values within every set of synthetic values. Nevertheless, test cases often require a more specific distribution of values, e.g., specific types of tax records, certain pre-defined vehicle data, or synthetic test persons of age groups defined by the test cases.

To allow synthetic data generation based on a specific pre-defined distribution of values within the ranges of all possible values of certain properties, we split the rules for these properties into equivalence classes. Every equivalence class represents a certain tax record, a vehicle data class, an age group, or any other attribute where the value is directly associated with a test case or another requirement of the user. We use conditional logic and pre-defined "helper values" and we define in the schema to which of the equivalence classes a synthetic data value that is generated for a specific property must belong. The creation of equivalence classes is currently a manual process, with the perspective of automating it in the future.

The output of the *Assign counts* activity is the schema with rules and counts.

The *Assign counts* activity can be summarized as follows:

```
#Assign counts
SET path to {properties with rules}
    FOR each property in properties with rules
        IF property type is object
            SPLIT description to equivalence classes
            ADD helper_id to every equivalence class
            ASSIGN count for every equivalence class as
            helper_id value
        ELSE
            SKIP
        END IF
    END FOR
RETURN schema
```

3.2 Schema-based synthetic data generation

This Subsection describes the necessary inputs for the schema-based synthetic data generation step of our approach, as well as the activities and outputs of each activity of the schema-based synthetic data generation step.

Generate synthetic data candidates The *Generate synthetic data candidates* activity aims to generate a population of candidate sets, with synthetic values for every required property in the schema. The generated synthetic values comply with the rules defined for respective properties as well as with the counts within every equivalence class of the rule for one property, if the equivalence classes and counts are defined. The *Generate synthetic data candidates* activity requires the user to manually enter the size of the population. It then takes the schema as input and uses a LLM-based synthetic data generator to generate the requested amount of synthetic data candidate sets. The gpt-4.1 from OpenAI is used to generate the synthetic data candidates. The value of the model temperature parameter is set to 0 to minimize the creativity of the model and to force it to explicitly follow the schema.

The output of the *Generate synthetic data candidates* activity is a JSON file that includes the generated synthetic data candidates.

Run sanity check The *Run sanity check* activity takes the JSON file with generated synthetic data candidates as input and validates it against the schema. The goal is this activity is to check that all required values exist. In other words, this activity mostly checks the performance of the LLM and identifies its failure to generate all requested synthetic data values.

The sanity check is performed automatically by a dedicated Python program. If no errors are found by the program, no manual intervention is necessary from the user. Should the sanity check identify errors in the generated data candidates, the user will receive the error report, which then needs to be investigated manually.

The output of the *Run sanity check* activity is the sanity check report, together with the error report, if applicable.

Solve constraints The *Solve constraints* activity aims to identify which candidate values are not allowed across properties and to correct these values. It takes the JSON file with the generated synthetic data candidates and the file with constraints as input and uses the gpt-4.1 from OpenAI as the constraint solver.

In addition to an LLM-based constraint solver, we have previously experimented with the Z3 solver from Microsoft Research[11]. While the Z3 showed more consistent results in 2024, the results of the Z3 and the gpt-4.1 were already similar in our context by the second half of 2025. Considering the vast

[11] https://www.microsoft.com/en-us/research/project/z3-3/

and continuous advancement of LLMs and the complexity of setting up the Z3 solver correctly for every individual set of data candidates and constraints, a LLM-based constraint solver is better suited for our context.

The output of the *Solve constraints* activity is either a statement "No incompliances found", or a JSON file where the generated synthetic data candidates with values that were identified as not compliant with the constraints are corrected. If the constraint solver outputs corrected synthetic data candidates, the respective candidates are replaced in the JSON file that includes all generated synthetic data candidates. This creates a final JSON file including the generated synthetic data as output of the Schema-based synthetic data generation step.

4 Validation

This Section describes the validation process of our PoC, defines the RQ, and validation metrics for answering the RQ.

To validate the PoC, we set up a playground with an application that uses data from three different Estonian e-government data services as input. The application served as our System Under Test (SUT). To emulate the real-life test data ordering process described in [2], we wrote test cases based on the functional requirements of our SUT, as well as synthetic test data ordering requests for all three data service providers to enable the execution of our test cases. We used the PoC to generate the requested synthetic test data.

Figure 3 shows the architecture of the playground. MySQL[12] was used to build the Synthetic Data Digital Twin (SDDT). The generated synthetic test data was inserted into the MySQL database. For each of the three data services, a mapping view was created to combine data from different tables into one dataset that matches the structure of the respective data service. MySQL REST Service (MRS)[13] was used to create the REST services that mimicked the request- and response structures of the three actual data services. We built the API with FastAPI[14], using the uvicorn[15] web server. The SUT was an executable Java application.

To explore some variations that are likely to occur in real-life-like settings, we designed three slightly different experiments.

Ex1: The SUT uses data from three different data services. The generated synthetic test data does not have to be consistent between the three data services.

Ex2: The SUT uses data from three different data services. One set of the generated synthetic test data has to be consistent between two of the three data services.

[12] https://www.mysql.com/
[13] https://dev.mysql.com/doc/dev/mysql-rest-service/latest
[14] https://fastapi.tiangolo.com/
[15] https://uvicorn.dev/

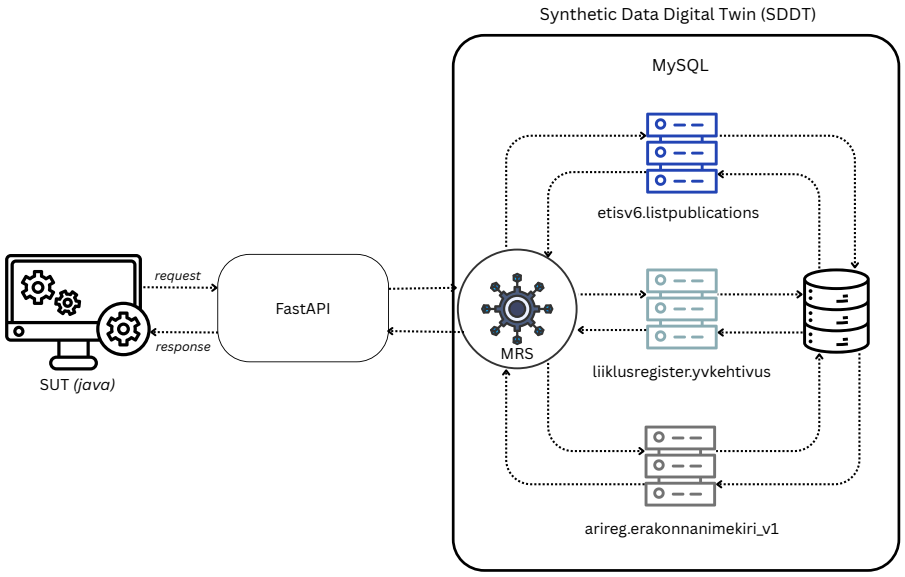

Fig. 3. Architecture of the playground.

Ex3: The SUT uses data from three different data services. One set of the generated synthetic test data has to be consistent between two of the three data services. The PoC can use only LLMs that store data in the EU.

We have defined the following RQ to be answered based on experiment results:

Does the suggested synthetic data generation approach support the ad hoc generation of synthetic test data in real-life-like settings?

4.1 Validation metrics

We have defined the following binary metrics to answer our RQ:

M1: Every requested synthetic test data is generated.

M2: The quality of generated synthetic test data enables the execution of all test cases on the SUT.

M3: Changing the LLM in the PoC does not affect the results of M1 or M2.

5 Results

This Section describes the validation results of our PoC and provides the answer to our RQ.

5.1 Results of Ex1

Every test case required a response with synthetic test data from all three data services. We were able to generate all the requested synthetic test data with our PoC. The generated synthetic test data did not cause any false negatives while executing the test cases.

5.2 Results of Ex2

Every test case required a response with synthetic test data from all three data services. One test case required that the value of an element in one data service would be the same as the value of an element in another data service. With our PoC, we were not able to generate test data for one test case, where the synthetic test data needed to be consistent between two of the three data services. We were able to generate all other requested synthetic test data. The generated synthetic test data did not cause any false negatives while executing the test cases.

5.3 Results of Ex3

Every test case required a response with synthetic test data from all three data services. One test case required that the value of an element in one data service would be the same as the value of an element in another data service. The mistral-embed embedding model and mistral-medium-latest from Mistral AI[16] were used instead of OpenAI's text-embedding-3-large embedding model and gpt-4.1. With our PoC, we were not able to generate test data for one test case, where the synthetic test data needed to be consistent between two of the three data services. We were able to generate all other requested synthetic test data. The generated synthetic test data did not cause any false negatives while executing the test cases.

Answer to the RQ: The suggested synthetic data generation approach supports the ad hoc generation of synthetic test data in real-life-like settings for simpler requests that don´t require data consistency between different data services.

Table 1 summarizes the experiment results by stating where a validation metric was achieved ("Yes"), where a validation metric was not achieved ("No"), and where a metric was not applicable ("N/A"). We publicly share the experiment reports with the PoC code and the generated synthetic test data.[17]

[16] https://mistral.ai/

[17] Data availability: https://doi.org/10.6084/m9.figshare.31699558

Table 1. Validation results

Validation metric	**Ex1**	**Ex2**	**Ex3**
M1 - test data complete	Yes	No	No
M2 - test data correct	Yes	Yes	Yes
M3 - LLM effect	N/A	N/A	Yes

6 Discussion

This study provides the first validation results for the PoC of a novel synthetic data generation approach that not only generates data but also aims to automate the underlying rule and constraint definition without accessing real-life raw data. The results of Ex1 demonstrate that the approach can effectively generate synthetic test data for test cases where no consistency between synthetic data received from different data services is required. This shows that the approach can potentially already be a useful alternative to the currently used manual test data ordering process, as described in [2].

The results of Ex2 point out that our approach has difficulties with preserving relations between different data services. More specifically, the LLMs struggle with some of the conditional logic used in the *Assign counts* activity, which is described in Section 3. Despite our investigation into this matter, which included the revision of the execution logs of OpenAI, searching for relevant literature, and consulting the LLMs themselves, we were not able to fully identify why conditional logic structured in the same way was ignored for one property, and followed for another property. We conclude that although LLMs are already great at reading JSON, it is not sufficient to simply include conditional logic in the schema. Although in our experiments, it prevented us from generating synthetic test data that is consistent between two data services, an unexpected failure to understand the conditional logic in a Schema may also cause failures in generating synthetic test data that meets the requirements within one data service.

For Ex3, we replaced the embedding model and LLM from the OpenAI product family with the embedding model and LLM of the EU-based Mistral AI. Changing the embedding model and LLM did not affect achieving the metrics: both LLMs were able to generate synthetic test data for most test cases, and both LLMs struggled with conditional logic in the Schema, when generating synthetic test data where consistency between data from two data services had to be achieved. This shows that the approach is generally robust enough that it can be operated with different embedding models and LLMs. Moreover, the POs that are described in Section 3 are designed to be so small and simple that they do not necessarily require very large or expensive LLMs.

There are, nevertheless, important limitations that need to be addressed in our future work. As the current PoC still requires human intervention in several activities, we will continue with automating the approach. We will implement a chain of additional POs for the LLM to automatically extract client input from

test data queries and translate those into data taxonomies and ontologies. These will be used respectively in the generation of synthetic data candidates and in solving constraints with the aim of increasing the alignment of the generated synthetic data with real-world patterns. Since we learned from this study that using conditional logic in the Schema is not a reliable solution, we will add the data generation instructions that were previously captured in conditional logic to property descriptions instead. We will also find an alternative for the Python Zeep SOAP Client that sometimes fails to parse WSDL documents due to namespace issues and requires WSDL documents to be patched manually. When developing the alternative, we will aim for a solution that can be used with WSDL, as well as OpenAPI descriptions. The improved approach requires evaluation in actual real-life settings, where e-government services are tested.

7 Conclusion

Testing of e-government services that rely on realistic synthetic test data that can be used as input from data received from data services. We developed the PoC of a rule-based synthetic data generation approach that, in addition to generating the synthetic data itself, also creates the rules and constraints that are the basis for synthetic data generation. Our approach does not require direct access to real-life raw data. We validated the PoC of our approach in a playground that simulated a real-life-like situation with an SUT requesting test data from three different data services. We found that although our approach has potential and could theoretically already be used in simpler cases, the PoC struggles with generating synthetic test data that is compatible across different data services. Our future work focuses on removing the identified limitations and evaluating the improved approach in actual real-life settings.

Acknowledgments. The research reported in this paper has been partly funded by BMIMI, BMWET, and the State of Upper Austria in the frame of the SCCH competence center INTEGRATE (FFG grant no. 892418), in the COMET (Competence Centers for Excellent Technologies) Programme managed by Austrian Research Promotion Agency FFG. Daniel Rodriguez is supported by JCLM project SBPLY/24/180225/000143 and TSI-100920-2023-1.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Anthes, G.: Estonia: a model for e-government. Communications of the ACM 58(6), 18–20 (2015). https://doi.org/10.1145/2754951
2. Tammisto, M.-A., Ramler, R., Shah, F.A., Pfahl, D.: Obtaining Test Data in the Estonian E-Government System: Challenges and Improvement Potential. In: Scanniello, G., Lenarduzzi, V., Romano, S., Vegas, S., Francese, R. (eds) Product-Focused Software Process Improvement. Industry, Doctoral-Symposium, Tutorial,

and Workshop Papers. PROFES 2025. Lecture Notes in Computer Science, vol 16362. Springer, Cham. (2026) https://doi.org/10.1007/978-3-032-12092-2_1
3. Tammisto, M.-A., Ali Shah, F., Rodriguez, D., Pfahl, D.: The Challenge of Generating and Evolving Real-Life Like Synthetic Test Data Without Accessing Real-World Raw Data—A Systematic Review. Expert Systems 42(12) (2025). https://doi.org/10.1111/exsy.70164
4. Baresi, L. and Miraz, M.: TestFul: automatic unit-test generation for Java classes. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10). Association for Computing Machinery, New York, NY, USA, 281–284. (2010) https://doi.org/10.1145/1810295.1810353
5. Wang, S., Shrestha, N., Kucheri Subburaman, A., Wang, J., Wei, M., Nagappan, N.: Automatic Unit Test Generation for Machine Learning Libraries: How Far Are We?. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1548–1560. IEEE (2021).
6. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: WS-TAXI: A WSDL-based Testing Tool for Web Services, 2009 International Conference on Software Testing Verification and Validation, Denver, CO, USA, 2009, pp. 326-335, doi: 10.1109/ICST.2009.28.
7. Ed-douibi, H., Cánovas Izquierdo, J. L., Cabot, J.: Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach, IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), Stockholm, Sweden, 2018, pp. 181-190, (2018) doi: 10.1109/EDOC.2018.00031.
8. Fuchs, A, Kuchen, H.: Test-case generation for web-service clients. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18). Association for Computing Machinery, New York, NY, USA, 1518–1527. (2018) https://doi.org/10.1145/3167132.3167294
9. Xue, Z., Li, L., Tian, S., Chen, X., Li, P., Chen, L., Jiang, T., Zhang, M.: LLM4Fin: Fully Automating LLM-Powered Test Case Generation for FinTech Software Acceptance Testing. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 1643–1655. (2024) https://doi.org/10.1145/3650212.3680388
10. Korraprolu, B. R., Pinninti, P., Raghu Reddy, Y.: Test Case Generation for Requirements in Natural Language - An LLM Comparison Study. In Proceedings of the 18th Innovations in Software Engineering Conference (ISEC '25). Association for Computing Machinery, New York, NY, USA, Article 9, 1–5. (2025) https://doi.org/10.1145/3717383.3717389
11. Smith, B., Troynikov, A.: Evaluating Chunking Strategies for Retrieval. Chroma Technical Report, Chroma. (2024) https://research.trychroma.com/evaluating-chunking
12. Wu, T., Terry, M., Jun Cai, C.: AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22). Association for Computing Machinery, New York, NY, USA, Article 385, 1–22. (2022) https://doi.org/10.1145/3491102.3517582
13. Arabzadeh, N., Chen, Z., Petroni, F., Siciliano, F., Silvestri, F., Trappolini, G.: IR-RAG @SIGIR25: The Second Edition of the Workshop on Information Retrieval's Role in RAG Systems. In Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '25). Association for Computing Machinery, New York, NY, USA, 4168–4171.(2025) https://doi.org/10.1145/3726302.3730362