Give me some REST: A Controlled Experiment to Study Effects and Perception of Model-Driven Engineering with a Domain-Specific Language

Maximilian Schiedermeier
schiedermeier.maximilian@uqam.ca
Université du Québec à Montréal
McGill University
Montréal, Canada

Jörg Kienzle
joerg.kienzle@{uma.es/mcgill.ca}
ITIS Software, University of Málaga /
McGill University
Málaga, Spain / Montréal, Canada

Bettina Kemme
kemme@cs.mcgill.ca
McGill University
Montréal, Canada

Abstract

Domain-Specific Languages (DSLs) are an efficient means to counter accidental complexity and are therefore a key technology for Model-Driven Engineering (MDE). Despite DSLs' potential, there is a lack of empirical research regarding the practical effects and developer perception of DSL-driven tools. In this paper, we present a controlled experiment with 28 participants around a previously developed DSL-based toolchain, which assists the migration of legacy software to REST. A direct comparison of developer performance for a) "*DSL toolchain*" and b) "*classic manual software migration*" allowed for analysis, quantification of effects and developer perception, as well as reasoning on general advantages, and DSL-related challenges. In certain cases, we measured a significant correlation between toolchain use and performance gains for developers. Detailed analysis of developer activities suggests the DSL toolchain alleviates tasks which show error-prone or time-consuming in the manual alternative. We then extracted acceptance-hindering factors from participant feedback and derived a series of recommendations for MDE practitioners who seek to develop DSL-based tools.

CCS Concepts

• **Software and its engineering** → **Domain specific languages**; • **Information systems** → **RESTful web services**.

Keywords

DSL, MDE, REST, controlled experiment, empirical study, software quality

ACM Reference Format:
Maximilian Schiedermeier, Jörg Kienzle, and Bettina Kemme. 2024. Give me some REST: A Controlled Experiment to Study Effects and Perception of Model-Driven Engineering with a Domain-Specific Language. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640310.3674080

1 Motivation

Domain Specific Modelling languages have been used in computing since the early days [42]. By reifying concepts of the problem domain in the language, DSLs mitigate accidental complexity [1] and consequently, DSLs are often used in combination with general purpose languages (GPLs) to develop larger software systems [14]. Although the integration of DLSs by generating code and then combining it with the remainder of the code base and software system is common [22, 26, 47], a systematic integration with GPL-based MDE is challenging: existing toolchains, i.e. series of GPL model transformations and refinements, cannot readily consume DSL languages or compose models not expressed in the same language [24].

To overcome this challenge, we previously proposed the FIDDLR framework [38], which defines clear, minimally-invasive tasks to integrate a DSL with an existing MDE pipeline, maximizing reusing of existing MDE tools and model transformations. We reasoned the framework renders the creation of DSL-based tool chains more attainable. We illustrated our approach on a representative and industrially relevant [9, 34] engineering task, namely, the migration of legacy code to cloud-ready services [15, 31]. The presented approach involves extracting and exposing legacy functionality through a new REST interface [11, 12].

Our previous publication showed quantitative evidence that a DSL+MDE, i.e. a DSL toolchain, requires less developer effort than a manual migration[1], but did not provide empirical evidence the approach provides benefits in practice. As such it remains unclear whether such DSL toolchains are a viable concept with measurable and substantial benefits compared to a manual code migration. The shortness of empirical evidence is an acknowledged ongoing issue in the MDE community [4, 18, 19, 29]. Therefore, we address this issue by reporting on a controlled experiment with 28 software developers to **test the hypothesis of a beneficial DSL toolchain effect and better understand the impact of using a DSL toolchain instead of manual code modifications**. All the data from our study is available publicly [2]. Similar to related SE

[1]We previously compared the number of required modeling actions to the number of lines of code that had to be added, removed or modified.
[2]*Data Availability Statement:* Our research was conducted with great care for transparency and reproducibility. We have prepared a browser-navigable replication package at https://m5c.github.io/RestifyReplicationPackage/, as well as an archived snapshot [37] with all raw data (except screen recordings), participant source code, model submissions, and CSV files with transcripts of all collected footage. The package includes time measurements, participant feedback, and observations from 72 hours of video footage. The bundle also contains a Jupyter notebook, including docker support for convenient replication of test results. We also provide source code and documentation for all tools implemented for the realization of the study, as well as recruitment material.

work [8], we measured engineering efficiency based on two constructs: "time to task completion" and "correctness of the outcome". The study allowed us to a) measure a beneficial effect of a DSL-based toolchain extension on software practitioners for the target engineering task, b) obtain qualitative observations to better understand how the suggested DSL extension contributes to the target task, and c) investigate developer bias and derive recommendations to improve developer acceptance of DSL toolchains.

The remaining structure of this article strongly orients on the reporting recommendations by Wohlin et al. [49] and Jedlitschka & Pfahl [20] for empirical software engineering research. The paper concludes with a recapitulation of the experiment insights, with emphasis on the inferred recommendations to further improve the acceptance of DSL toolchains.

2 Background and Related Work

2.1 Background

In the first part of this section, we first briefly recall the background on controlled experiments in software engineering, and then outline a DSL and associated MDE-based toolchain for migrating legacy applications to offer services following the REST style [12]. In the second part, we present related work on controlled experiments involving MDE and DSLs.

2.1.1 Controlled Experiments in SE. Empirical research originated in a medical context, e.g., the testing of drug treatments on patients, to gain insights on the drug's effects [5, 7]. The established methodologies have been adopted by Computer Science researchers, leading to a framework for the correct application of empirical research methods in our field [40]. Best reporting practices include using the correct terminology and a template structure [20, 35, 49].

Crossover Layouts. The design chosen for our study is a variant of the *crossover layout* [7], where a sequence of treatments and observations are conducted on the same subject (human participant). Individual pairs of treatment and observation are also called experiment *periods*. Cross-over layouts, as repeated measures experiments, have the advantage of producing multiple samples per subject. However, the layout also introduces new challenges. Measurements of the same variable on the same subject are not independent, which must be taken into account when selecting appropriate statistical tests [43].

Carryover, Blocking Variables and Factorial Design. Repeated measures experiments also need to consider *carryover*, which is the unintentional measurement of the effects of a previous period's treatment. If, e.g., first a new drug and afterwards a placebo were used as treatments on the same subject, the observation for the placebo period can be influenced by the lasting effects of the previous drug treatment.

For Software Engineering experiments, a common strategy to overcome this issue is the introduction of a blocking variable [5], that is, a contextual *object* that changes between periods to mitigate carryover. For instance, Ceccato et al. [6] investigated two software obfuscation strategies (treatments). Each strategy was applied to a different software application (object), so that the knowledge obtained in the first period did not influence the second period. A *factorial crossover design* then refers to forming sequences based on all experiment factors, e.g. treatment, object and order.

Statistical Analysis. Typically, an experiment seeks to determine relationships between explanatory variables (e.g., taking the tested drug), and construct variables (e.g., capturing health improvements). To do so, a *Null Hypothesis*, assuming no correlation, is formulated. A statistical test can then, based on the presented data, reject the null hypothesis, meaning there is a significant correlation. Only rejection of a Null Hypothesis is a conclusive result; failure to reject does in turn not mean that the hypothesis is true [48].

Factorial crossover layouts showcase multiple factors or explanatory variables. It is therefore insufficient to only investigate the assumed most plausible one - even if a statistically significant correlation is observed, other factors might be even more relevant. For this layout, linear models are a recommended analysis start, for they give a preliminary estimation of how individual factors contribute to the observed effects [43].

2.1.2 Migration to REST. REST interfaces constitute a layer of abstraction that conceals functionality behind hierarchically structured resources, accessible over HTTP via CRUD operations (Create, Read, Update, Delete) [11].

Especially the appearance of Micro-service architectures has made REST-based service interfaces a popular approach, and as a result, the migration of legacy code to a RESTful service is an industrially relevant, yet challenging task [3].

To achieve such migration, one has to conceptually a) design a hierarchical resource tree, b) assign operations on individual resources to appropriate HTTP request types, and finally c) map assigned operations to methods in the legacy application [12]. In practice, platforms that support REST, e.g., the Spring framework [44], introduce additional development complexity. Steps a), b) and c) are done implicitly through code annotations, attached to the signatures of legacy code methods. Furthermore, non-trivial configuration files and compilation dependencies have to be specified.

DSL and MDE Support for REST. Because the REST resource tree crosscuts the module structure of the application functionality, REST annotations often end up scattered over the code. It is unclear how the lack of a global perspective on the produced API structure and the distraction caused by technical details affect the developer.

Consequently, several DSLs have been proposed for REST that accurately capture the nature of resources and CRUD operations with tailored concepts: The *RESTful API Modeling Language* (RAML) [32] and the *Web Application Description Language* (WADL) [13] are both textual and low-level, OAS/Swagger [41] puts the focus on a machine parsable JSON format for stub code generation, and finally, the *Web Resource Modeling Language* (WRML) [30] raises the level of abstraction with a resource tree representation. Yet, none of these DSLs were crafted specifically for the migration of legacy software. Similarly, the semi-automated, and model-driven migration of legacy applications to REST has been discussed before [36], however not setting on a REST- or migration-oriented DSL.

For this study we therefore used *ResTL*, a graphical DSL specifically designed for modelling the specification of hierarchically arranged resources [38]. ResTL was integrated with an MDE toolchain and TouchCORE [23], an open-source academic modelling tool for concern-oriented software design. The tool allows the developer to specify mappings of the REST endpoints to existing functionality. The MDE pipeline provided by TouchCORE, which includes model weavers and code generators, takes care of generating the REST-related source code. As a result, the developer does not have to deal with the scattering of annotations throughout the original code base, as well as from configuring the chosen REST implementation framework. More details on the DSL+MDE process and how it is experienced by the ResTL user are covered in Section 3.2.

2.2 Related Work

DSLs have been shown to increase productivity, in particular because they raise the level of abstraction away from implementation details, and because of the gain in correctness and speed thanks to code generation [16, 47]. For example, [22] reports the results of an experiment in which a template-based approach and a DSL approach to software generation were compared. Several subjects were monitored while performing several development and maintenance tasks using DSL technology and alternative template technology. Flexibility, productivity, reliability, and usability were measured. The DSL approach scored better on all counts. More recently, [26] ran a controlled experiment where developers had to perform a program comprehension task either using a DSL or using the corresponding API in a GPL. Results showed that developers are significantly more accurate and efficient when using DSLs than when using GPLs. Finally, [2] presents two scientific applications in which the use of DSLs added value by raising the level of abstraction and enabling project-level reasoning. However, no controlled experiment was conducted in this case.

At an OOPSLA panel in 2008 on the advantages and problems associated with DSLs, academic and industrial researchers as well as commercial tool vendors identified several difficulties in working with DSLs [17]. While the actual development of the DSL itself is challenging, several panellists also mentioned that they witnessed how developers in both academia and industry are generally reluctant to learn new languages.

Over the past years, several research groups have contributed to fostering a better understanding of various aspects of DSLs, MDE, or toolchain effects. [19] and [18] set on thorough qualitative and quantitative methods to interview professional practitioners and better understand the various social, technical and organizational factors that influence the success or failure of MDE techniques in SE industry. The authors also condensed their observations to a series of recommendations for the MDE community, as orientation for when and how to best apply MDE. Notably, the authors found that the widely assumed merits of code generation are less relevant to practitioners than other merits, e.g. contribution to architectural documentation.

Several research groups used controlled experiments to better quantify and understand the effects of individual factors. [29] conducted a crossover layout study to investigate how MDE contributes to maintainability tasks in a web application context in comparison to a manual approach. The authors measure the correctness of performed developer tasks and the time required for completion and observe a significant performance gain for the MDE approach. Similarly, [8] describes a controlled experiment in crossover layout to compare DSL vs. GPL behavioural modelling approaches in a video game context. The authors then studied gains concerning task correctness and efficiency metrics and related them to subjective developer feedback on perceived model usefulness. A side insight of the study is that independent of generally high model correctness, participants regard models as sketches rather than programs. [27] describes an experiment to assess whether the reduced accidental complexity associated with DSLs impacts developer understanding of API usage in direct comparison to a GPL counterpart. The authors set on *correctness* and *efficiency* (time) constructs across three experiments and conclude significant improvements for both metrics when using the DSL approach. Finally, controlled experiments have also been conducted in the context of REST. [39] describes an experiment to compare developer understanding of REST API usage, once more assessed using *correctness* and *efficiency* (time) constructs.

Despite these significant contributions, no research has been conducted to measure and understand the effects of a DSL-extended tool chain. Especially concerning migrating legacy software to REST, there is no related empirical work, to better understand tool chain effects in direct comparison to a manual migration. A better understanding in this context seems especially important, given the migration of legacy software is an ongoing, unresolved and industrially highly relevant challenge.

3 Experimental Design

We applied a *two-treatment factorial crossover design with a level-two blocking variable*. In simple terms, this layout allows for a fair comparison of two alternative software migration techniques, where every participant produces two samples: One from a software migration using a ResTL DSL with an associated MDE toolchain, and the other from the manual migration of the legacy source code using an IDE. Table 1 illustrates the resulting study design. The interest of this layout is to support a comparison of the two software migration techniques, concerning the quality of the outcome and time needed for the migration process. We set on two variables for this comparison: correctness of the migrated software, by a *test pass-rate* construct, and migration efficiency via *task solve time* from start to end of the migration process. We assess whether migration methodology (treatment) or task order correlates to a significant performance gain for the dependent variables. We use linear models to estimate the relevance of methodology and order as experiment factors (explanatory variables) and estimate effect size by comparing pass rates and task-solve-time distributions. Additionally, the layout allows us to correlate variable measurements to qualitative observations, in support of a better understanding of DSL toolchain effects.

3.1 Layout Details

While each study participant applies both techniques (treatments), no participant ever works twice on the same legacy software (object), and we also do not directly compare samples obtained from the same subject. Alternating the legacy software (object) between the first and second migration task serves as a two-level blocking variable, where "*two level*" indicates the two software alternatives.

We want to consider as many factors as possible, therefore the *order* of migration technique applied (treatment) and legacy software

Experimental Design	**Task 1**: *Treatment + Object*	**Task 2**: *Treatment + Object*
Group I (Red)	DSL Migration + BookStore	Manual Migration + Xox
Group II (Green)	Manual Migration + BookStore	DSL Migration + Xox
Group III (Blue)	DSL Migration + Xox	Manual Migration + BookStore
Group IV (Yellow)	Manual Migration + Xox	DSL Migration + BookStore

Table 1: Two-Treatment Factorial Crossover Design

worked with (object) is not identical for all participants (subjects). Each combination of migration technique (treatment) and legacy application to migrate (object) forms a study **task**. A study period consists of a task performed and the corresponding data collection (observation). The four resulting combinations of tasks (combination of *object* and *treatment*) form four experiment sequences.

For fairness we allocate the same number of participants to each of the four sequences, i.e. we divide our population into four equally sized **groups** (labelled I-IV and corresponding colour).

3.2 Treatments

Migration using a DSL toolchain: The DSL-oriented approach does not require any manual coding. The TouchCORE modelling tool can load and interpret legacy sources to extract and visualize structural program information as class diagrams. Using the specific ResTL DSL editor, the user graphically designs the desired REST resource tree and CRUD operations. Afterwards, they graphically map those resource operations to individual methods of the legacy code. Figure 1 illustrates how the tool is perceived by a user. No additional action is required from the developer, for the tool applies model transformation and composition techniques to generate source code and a build configuration file. The resulting service is compiled and deployed with a `maven` command.

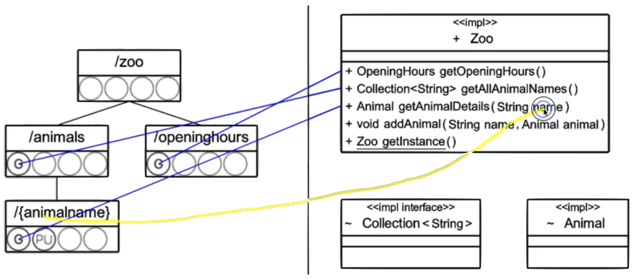

/zoo
/animals
/openinghours
/{animalname}
G
G
G
PU
<<impl>>
+ Zoo
+ OpeningHours getOpeningHours()
+ Collection<String> getAllAnimalNames()
+ Animal getAnimalDetails(String name)
+ void addAnimal(String name, Animal animal)
+ Zoo getInstance()
<<impl interface>>
~ Collection<String>
<<impl>>
~ Animal

Figure 1: Capture of the DSL toolchain in action: The ResTL DSL (left) represents a REST resource tree. Blue lines are mappings of CRUD operations (circles) on operations in a class diagram. The yellow line is the user establishing a mapping of a placeholder resource to an operation parameter.

Manual Migration: Manual migration translates to modifying the legacy application's source code by hand. The developer loads the legacy sources in their Integrated Developer Environment (IDE) and then adds support for the REST technology of their choice. For the study, we restricted the technology choice to Spring Boot, which is a reasonable choice due to its widespread use. According to the *2018 JVM Ecosystem Report*, a survey based on 10,200 questionnaires, Spring Boot is the most popular Java web framework [28].

Regarding the IDE, participants were encouraged, but not mandated, to use the same IDE as used in the task illustration material, namely the IntelliJ IDEA. After modifying the build configuration to support Spring, the developer has to add REST-technology-specific annotations to the code to expose operations. Correct integration of Spring also requires writing some boilerplate code and adjustment of project configuration files.

3.3 Objects

Both legacy application objects were crafted as study objects for migration to REST. They are written in Java, have no JDK-external dependencies, include a simple maven build configuration, have 100% test coverage, and are extensively commented.

The BookStore: The first legacy application imitates the functionality of a common e-commerce scenario. API calls allow for indexing books by ISBN, maintaining reader feedback and accessing stock information for available book copies in stores. Migration of the BookStore requires the re-exposure of 12 API methods.

Xox: The second object is a re-implementation of the turn-based two-player game Tic-Tac-Toe (referred to as "Xox", paraphrasing the *X*es and *O*s on the board). The Xox API allows players to participate in parallel game instances, following the blackboard pattern. The implementation was created to match the semantic complexity of the BookStore while changing the domain. Migration of Xox to a RESTful service requires re-exposure of 8 API methods.

Regarding the source lines of code (SLOC) metric, the two applications are of comparable size. The BookStore codebase without comments amounts to 737 lines (including build system configuration), and the Xox equivalent amounts to 1018 lines. Although migrating Xox touches fewer methods, the transitive REST concepts and annotations needed for a successful migration are identical for both applications.

4 Execution

Recruitment and study execution were carried out from June to August 2022. We launched an extensive recruitment campaign reaching out to industrial engineers, engineering students and academic colleagues. We used various mailing lists, sent invitations directly, and recruited using a dedicated web page. All material and software described in this section are included in the replication package.

Recruitment contained a preliminary self-assessment form, which we used to filter participants who did not fulfil the minimum required technical proficiency. In total, we recruited 28 subjects from approximately equal shares of academic, student and industrial backgrounds.

We ran several pilot studies to ensure the two conversion tasks could be completed within two hours. Due to pandemic precautions, participants were allowed to use their own computer setup, with the option to access a lab-provided workstation as a fallback. Participation was rewarded with an Amazon gift card equivalent to approximately 75 USD in local currency. We did not perform a sample size estimation, because in our study the population size was limited by the available funding.

4.1 Group Allocation

Participant recruitment included filling out a self-assessment form regarding experiment-relevant skills. Applicants were asked to declare their proficiency for the *Spring* framework, the *Maven* build system, the MDE tool *TouchCORE*, *Command Line* usage, the *REST* paradigm, the *Singleton* pattern, and *Reflection* (i.e., the programming language concept). We provided a textual metric for the self-assessment, to bring some objectiveness to the declared level of proficiency, on a scale of one to five. Listing 1 illustrates the assessment by presenting the questions regarding the *Singleton* pattern:

We used the self-assessment forms not only to ensure sufficient proficiency for participation but also to counter group bias. While initial observations on the population are common in crossover studies [7], experiments with a large number of participants usually rely on randomization to balance groups [5]. While the total population size is in line with comparative controlled experiments [25], we were concerned about group bias, given only 7 participants per group resulting from the presented factorial layouts.

For the allocation of participants to individual groups, we therefore did not rely on randomization, but we applied a heuristic to ensure comparable skills between groups.

```
How much do you know about the singleton pattern?
1 [ ] I don't know what it is
2 [ ] I know what it is, but have never used it
3 [ ] I have already used it in one of my projects
4 [ ] I could verify a provided implementation
5 [ ] I could implement it right away from scratch
```

Listing 1: Recruitment Self Assessment Form Excerpt

Specifically, the heuristic aims at minimizing the maximum distance in average skill proficiency between groups (MiniMax). Figure 2 shows the outcome of the balancing process. Distributions of the same colour represent the four resulting groups for each of the seven assessed skills. We observe that groups are reasonably comparable on all assessed skills. However, it is important to retain that, despite efforts for objectivity, self-assessments are by nature subjective. The participants' factual skills may deviate from their stated skills.

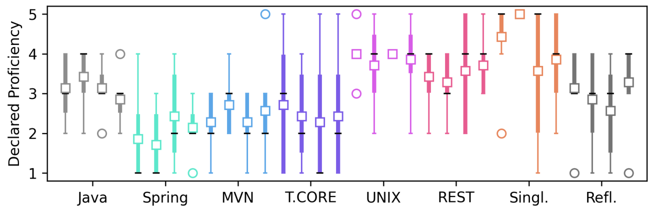


Figure 2: Skill Distributions Across Groups

4.2 Training

Both experiment tasks are preceded by a dedicated training phase, where participants are familiarized with the methodology to apply. The training material consists of a video, illustrating key steps of the migration task on a third, unrelated sample application. For the manual migration, the video repeats configuration file changes, and boilerplate code changes needed for migration, as well as the syntax of REST-specific annotations to decorate and expose existing legacy functionality. For the DSL tool chain migration, the video illustrates the usage of the TouchCORE software, how resource models can be created and linked to models extracted from the existing legacy functionality, and how code is generated. In both cases, the video is accompanied by a structured, textual summary of the main migration steps, providing the essential technical background. The training does not contain any information specific to the actual study objects. Videos and textual summaries are part of a sequential, easy-to-navigate website that guides participants through their study sequence.

4.3 Migration Instructions

The expected migration outcome is provided as a textual REST interface description. Participants are instructed to migrate the provided legacy software (object) to a RESTful service, corresponding to the provided description.
The textual format used as API description is identical for both tasks and was selected to avoid bias to either methodology, i.e., it neither contains full resource location paths, which could be copy-pasted directly into REST annotations nor does it provide a graphical visualization of the interface structure as a tree. It merely explains which operations on resources are expected to give access to what functionality. The subject must then determine how this translates to methodology and existing legacy functionality.

Additionally, the instructions contained source code, navigable JavaDoc and class diagrams illustrating the structure and functionality offered by the legacy application. For fairness, the same information was provided, regardless of the methodology used for a given migration task.

4.4 Data Collection

For each migration, we collected the participant-produced source code and recorded all their on-screen task activity.

- *Source code* is the final product of each migration, no matter the treatment. Although for the DSL toolchain, we requested the submission of all models, we generated the corresponding source code for comparability. In either case, the outcome includes a build system configuration, which allows compilation and local deployment of the produced RESTful service.
- *Screen recordings* cover all on-screen activity throughout the entire task, including the training period, i.e., from the moment of first familiarization with task-related techniques to submission of the migration outcome.

In addition to task-specific data, we also collected textual participant feedback after completion of both study tasks. We provided a structured template with questions on the perceived relative difficulty of the tasks, questions on the confidence in the correctness of the outcome, as well as a freely fillable text form to mention encountered difficulties and methodological preferences.

We analyzed source code and screen recordings concerning *correctness* of the produced software, and *efficiency* of the migration

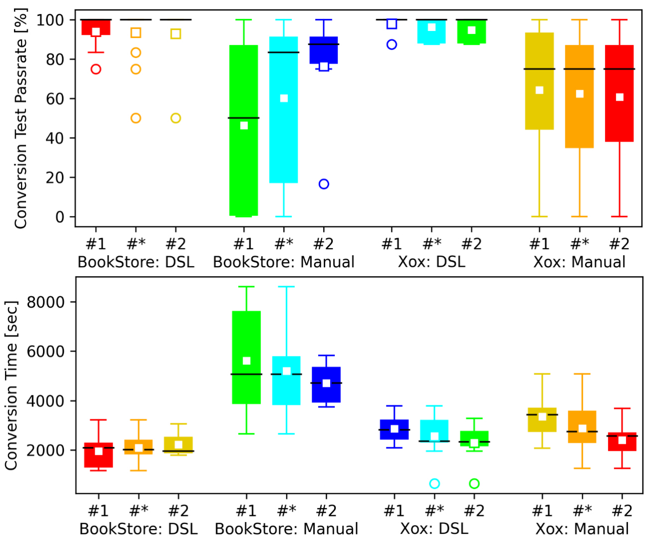


Figure 3: Distributions of *Pass Rates* and *Conversion Times*

	BS: DSL (#∗) (Orange)	BS: Manual (#∗) (Turquoise)	Xox: DSL (#∗) (Turquoise)	Xox: Manual (#∗) (Orange)
Q4: Max (Upper whisker)	100.0% / 3060s	100.0% / 8615s	100.0% / 3602s	100.0% / 5084s
Q3: 75% Quartile	100.0% / 2439s	91.7% / 5835s	100.0% / 3043s	87.5% / 3619s
Q2: Median (Black Bar)	100.0% / 2021s	83.3% / 5069s	100.0% / 2357s	75.0% / 2754s
Average (White Square)	93.5% / 2091s	60.3% / 5197s	96.2% / 2564s	62.5% / 2881s
Q1: 25% Quartile	100.0% / 1769s	16.7% / 3789s	87.5% / 2096s	34.4% / 2250s
Q0: Min (Lower Whisker)	100.0% / 1167s	0.0% / 2651s	87.5% / 1955s	0.0% / 1251s

Table 2: Distribution Quartiles, Disregarding Order (#∗)

process. Both constructs are a common choice to capture the efficiency of a software engineering activity [8].

- **Submission correctness** was assessed by testing submissions against the expected target REST interface. We provide a simple script to determine test results and the resulting ratio, as a normalized *test pass ratio* metric. Both sample applications were intentionally coded without persistence, which enables interdependent testing by simply restarting to reset the service state after every test. The only exception to this rule were "Get" requests applied to *verify effectiveness* of state-changing API requests, i.e. all "Put", "Post", "Delete" requests.
- **Migration efficiency** was determined by analyzing a total of 72 hours of screen recording and accurately distinguishing between the "task familiarization" and "task solving" phases. Regarding all subsequent statistical analysis, *time* strictly refers to task-solving activities, not task familiarization.[3] We allowed study discontinuation after a minimum effort of 1 hour per task (including training and material familiarization). However, only one participant aborted concerning this threshold. We therefore do not consider time a censored variable for the study.

Figure 3 depicts per-group distributions of our measurements for *correctness* and *time* constructs. Groups are represented using colour, the conversion technique, the application and task order are shown as labels ("#1" = *first* period, "#2" = *second* period, whereas "#*" shows the distribution of the two groups combined). Table 2 lists the corresponding numeric distribution data, combining samples of identical application (object) and migration technique (treatment).

5 Analysis

In this section, we present the methodology and results of our statistical analysis. We want to investigate **the hypothesis that the DSL toolchain provides a significant beneficial effect to software developers**, for the presented migration task. The analysis has two components, following the general recommendations for crossover experiments [43]. We apply linear regression models to assess the relative contribution of multiple experiment factors, concerning the correctness and efficiency observations (as opposed to investigating only *treatment*) (see subsection 5.1). The second component is a targeted pairwise comparison of pass-rate and migration time distributions, for when the same application is migrated with different methodologies (treatments). In the final step, we put the analysis results into relation and report effect sizes (see subsection 5.2). Note that all tests reported in this section can be conveniently replicated, just with a browser, using our replication package.

Outliers: We excluded the data of one participant (i.e., two produced projects and the corresponding two screen recordings) from our analysis. The performance measurements were extreme outliers (fastest migration, almost no tests passed), and investigation of the code and screen recording suggested the subject had no interest in training, following the instruction material or producing meaningful results.

Statistical Inference Corrections: With an increasing number of tests, there is a rising risk for type-I errors (false positive / falsely rejecting a hypothesis although it is true). Our analysis showcases three test families, we therefore apply the Holm-Bonferroni correction for the tests within each family. These families are 1) F-stat probability, used to assess the quality of our linear regression models, 2) testing the statistical significance of individual linear model variables, and 3) the pairwise comparison of distributions. The Holm-Bonferroni correction orders p-values within each family and then uses an adjusted correction on the standard $\alpha = 0.05$ threshold. For convenience, we highlight all p-values that are below their adjusted α level in bold (null hypothesis rejected).

5.1 General Linear Models

We created four general linear models (GLMs), all with migration methodology (*treatment*) and task *order* as explanatory variables. We then test the models' explanatory variables for statistical significance.

Note that this is a repeated measures experiment, with non-independent measurements (we have two samples per each participant). The GLMs are therefore designed to contain only one sample

[3]The original recordings cannot be released, to ensure participant anonymity. As we are notably interested in understanding tool chain effects, we provide transcripts of all recordings, listing all observations regarding participant behaviour, potential task deviations and observed technical issues.

Dependent Var.	App	Prob. F-Stat	Explanatory Var.	Coef.	Std.Error	p-value
Pass Rate	BS	0.018	(constant)	(53.90)	(9.94)	(74.41)
			(order)	(13.76)	(11.62)	(0.248)
			(treatment)	(32.67)	(11.62)	(0.010)
	Xox	**0.020**	constant	64.21	9.52	**0.000**
			order	-3.43	11.14	0.761
			treatment	33.79	11.14	**0.006**
Task Time	BS	**1.97e-05**	constant	5332	449.9	**0.000**
			order	-292	526.3	0.584
			treatment	-3095	526	**0.000**
	Xox	0.059	(constant)	(3263)	(279.8)	(0.000)
			(order)	(-762)	(327.4)	(0.029)
			(treatment	(-288)	(327.4)	(0.388)

Table 3: General Linear Model Results, four OLS Regressions

per participant, i.e. samples for one legacy application (*App*), resulting in a total of four GLMs. Table 3 lists the model results, based on an Ordinary Least Squares (OLS) regression, and the p-values for the statistical significance of each model variable. Note that each model also reports the quality of its regression (Prob. F-Stat). With the correction applied, only two models pass this quality check, i.e. only for two models we can reject the null hypothesis that the linear regression is not better than a flat model, with only constant. We therefore only interpret the results for these two models: *Pass Rate for Xox*, and *Task Time for BookStore*. The other two models should not be interpreted, Table 3 therefore shows their model values in parentheses.

The interpretation of the *Pass Rate for Xox*, and *Task Time for BookStore* models is as follows: In both cases, correlation is only confirmed (null hypothesis rejected) for *constant* and *treatment*. The absolute coefficient for *treatment* being of comparable magnitude as constant means according to this model the migration technique provides a measurable effect for the observed pass-rate and time measurement. The models do not provide conclusive evidence regarding the relevance of the *task order*.

5.2 Wilcoxon Rank Sum

We also investigated our initial hypothesis of an assumed beneficial effect of migration technique, by inspecting comparative performance distributions. To increase statistical power, we grouped observations by application (*object*), disregarding task *order*, and compared how pass-rate and time distributions evolve depending on the migration technique (*treatment*) applied. These comparisons correspond to four null hypotheses: Identical distributions concerning the measured migration time for BookStore, respectively Xox, and identical distributions concerning the measured test pass-rate distributions for BookStore, respectively Xox.

We applied the Wilcoxon Rank Sum test for these comparisons. This test was chosen because it is robust for small sample sizes, and does not make assumptions on the sample distribution [43] (we do not know the sample distribution, other than the preliminary boxplot visualization).

The p-values, in order of appearance of the above null hypothesis, are: **0.000016** (*time*, bookstore), 0.35 (*time*, xox), **0.00489** (*passrate* bookstore), **0.00389** (*passrate* xox). The corresponding interpretation is that for pass-rate measurements of BookStore and Xox, as well as for time measurements of the BookStore, the migration technique (treatment) causes a significant effect.

In the case of pass rate for Xox and time for the BookStore, the GLM models and Wilcoxon results draw a consistent picture, identifying a significant effect on the migration technique applied.[4] We further discuss the Wilcoxon test result for task-time of Xox in the next Section 6.

5.3 Effect Sizes

The GLM model tests and Wilcoxon Rank Sum tests attested the significance of the migration technique in several cases. For the reporting of effect sizes, we compare the evolving of pass-rate and migration time distributions. Table 2 indicates the numeric quartile information excluding outliers, except for *Average*) for the orange and turquoise boxplots of Figure 3, i.e., the distributions for all samples of participants working on the same application with the same migration technique.

We observe a relative average improvement whenever the DSL technique is applied instead of a manual code migration. The numeric differences in distribution average are as follows: For the BookStore an average pass rate improvement from 60.3% to 93.5% (33.2% higher pass rate), at an average task speedup from 5197 to 2091 seconds (3106 seconds faster); for Xox an average pass rate improvement from 62.5% to 96.2% (34.3% higher pass rate), at an average task speedup from 2881 to 2564 seconds (317 seconds faster). In all cases, we also observe a lower variability of the distributions whenever the DSL technique is applied.

6 Discussion

We now discuss how the quantitative results of our statistical analysis relate to contextual knowledge and qualitative observations. In detail, we first discuss individual traits of either migration technique, to explain either offset or absence of offset in concerning developer performance 6.1. Afterwards, we assess developer feedback and discuss an observed discrepancy between developer perception and measured performance 6.2. The section concludes with several critical factors for the practical success of DSL toolchains. We present it in the form of a lessons-learned list as a reference for the crafting of future DSL toolchains 6.3.

6.1 Understanding the Offsets

Lower Test Pass Rate for Manual Conversion: Common to both applications, is a lower test pass-rate, and higher variance in results, of the manual migration approach, i.e. based on our experiment data, switching to a DSL toolchain-guided migration results in consistently higher test pass-rates. We found several explanations for this observation.

- **Annotation syntax**: Analyzed screen recordings revealed that the participants had recurring difficulties with Spring's annotation syntax. We frequently observed participants confusing `@PutMapping` with `@PostMapping`, or forgetting an intermediate resource in the URL string. This error type barely occurred with participants using the DSL editor counterpart, which could be

[4]If interpreted, the GLM model (which barely surpasses the corrected f-stat threshold) would be likewise consistent with the Wilcoxon Rank Sum test results. Note that failure to reject a null hypothesis is not a conclusive result and statistical inference corrections bear a risk to introduce type-II errors (failure to reject a null hypothesis that is actually true).

an indicator for the visual approach to be more intuitive. The presumption is coherent to the majority of participants also reporting the DSL approach as more intuitive in their feedback form (see Figure 4).

- **HTTP parameter types**: We noticed recurring issues related to the mapping of request parameters. We observed several subjects running online searches for the correct annotation syntax and subsequently confusing various parameter types (Spring showcases similar-looking annotations for different HTTP parameter types). The study (and consequently also the training material) only includes resource (variable on resource path) parameters and HTTP-body parameters, but no HTTP-query parameters. Yet some participants mistakenly used the syntax for HTTP query parameters, which resulted in test failures.
- **Launch configuration**: Task instructions included a modification request for the build systems launcher entry, so the created REST service can be started from the command line. Four participants only tested their software in the IDE (using the green launch icon), rather than launching with the build system command. Ignoring this small mistake would have drastically changed the results: A service that is correctly coded, but misses a launcher configuration will still fail all tests. We therefore decided to correct the mistakes made in the launcher configuration[5]. Without the intervention, the observed pass rate distribution for the manual migration would have been lower than indicated in this report. Without our intervention, the observed offset compared to the DSL toolchain approach would have been larger.

Slower Manual Conversion for the BookStore. The statistical analysis shows a significant migration task speedup when the DSL approach was used for the BookStore. Interestingly no statistically significant effect is found for Xox. We first provide general reasons to explain a slower manual conversion, and then discuss why the observed effect is less important for Xox. [6]

- **Spring integration**: Screen recordings showed that configuration of the configuration *Maven* build system is a time consuming activity. In detail, manual migration required editing the project's `pom.xml` configuration file, adding a dependency statement to the Spring framework and updating the *Launcher* class reference. Although these changes are boilerplate activities, were detailed in the instructions and available as copy-paste-ready configuration snippets, many participants reported issues, and we likewise observed participants pasting provided snippets in the wrong place, and overlooking notably the launcher configuration.
- **IDE syntax highlighting**: In four cases, we observed participants struggling, associated with the IDE's syntax highlighting. Usually, the IDE automatically detects dependencies added to the `pom.xml` configuration file and automatically extends syntax highlighting to the additional dependencies. In four cases, we observed how participants correctly integrated spring dependencies, but their IDE did not automatically register changes. When participants then attempted to integrate Spring language constructs, they were bombarded with syntax errors. In all cases, the developers eventually worked their way around the issue with a manual configuration file reload.
- **Dependency injection**: We observed recurring issues related to Spring's dependency injection mechanism. The manual conversion commonly requires the incorporation of Spring's *Dependency Injection*s. In simple terms: if one class needs another, Spring annotations allow highlighting which dependency should be injected where. The interest is mainly to prevent the active claiming of dependencies and advocate coding against interfaces. In the case of BookStore, migration to REST implicitly requires correct application of dependency injection, to satisfy dependencies between the multiple controller classes, which embed the functionality to be exposed over REST. Although the concept is covered in the training material, 7 participant feedbacks forms explicitly mentioned difficulties experienced with dependency injection, and screen recordings show that even more participants were struggling. Screen recordings also show that participants persistently resolve this error by investing additional time.
  In the case of Xox, all functionality to expose over REST resides in a single class, hence there was no dependency injection to resolve. Consequently, we did not observe a slowdown for manual migrations of Xox. However, Spring applications beyond a minimal size all exhibit dependency injection. We thus think the offset measured for the BookStore reflects reality better than the one of Xox. The issue of Xox not requiring the use of dependency injection is further discussed in the threats to validity Section 7.

Attributing effects: We can to some extent draw a connection from the analyzed performance offsets and observed challenges to MDE concepts, as the main driver of our sample DSL toolchain. The DSL aims at directing the developer's attention away from syntax details, towards the task semantics. The observations of **annotation syntax** struggles and confusing **HTTP parameter types** in the manual approach are plausible representatives to explain differences in the resulting project correctness. Both issues do not occur in the DSL toolchain alternative.

We can also see the beneficial effects of code generators as standard MDE concepts. The observed delays associated with off-the-shelf build system modifications for **launch configuration** and **spring integration** are plausible factors for a more consistent and on average shorter migration duration, compared to a higher variability on the manual approach.

We can also associate the absence of tedious technical issues, i.e. the confusing **IDE syntax highlighting** due to undetected `pom.xml` changes, and resolving **dependency injection**, with MDE's general ambition to abstract technical intricacies away by efficiently integrating prepared reusable solutions (which effectively is the case, for the DSL toolchain internally reuses design models to automatically ensure correct framework integration).

Note that it is not obvious how to draw a line between DSL effects and standard MDE effects. Unless used for illustrative or sketching purposes, a DSL cannot be separated from its tailored MDE transformations. As such, it is not obvious how to attribute effects to individual toolchain components.

[5]The original sources in the replication package contain a marker, to indicate our manual modification.
[6]Note that in principle, migration *time* would have been a censored variable because participants were allowed to discontinue the study once the minimum time had passed. However, screen recordings revealed that participants continued until their project was at least compiled (except for the excluded outlier). The vast majority of participants also selectively tested REST access before submission.

Toolchain issues. : Statistical analysis, observations and discussion speak in favour of a beneficial DSL toolchain effect concerning test pass rates. However, regarding the migration time, we cannot conclude a beneficial effect as consistently as expected. We believe this can be explained by the academic nature of the toolchain implementation, i.e., the toolchain is not production-ready. It is possible that the beneficial effect would have been observed more consistently if an industrial toolchain implementation had been available.

- **Crashes**: About a third of participants experienced crashes of the toolchain. Although a dedicated measurement showed the time losses were minimal, we did not exclude crash times from our measurements to avoid any bias. Screen observations confirm that participants rapidly restore their work from checkpoints.
- **Navigation**: While all participants concluded the toolchain migration, we observed consistent issues with navigating the software. As the name suggests, TouchCORE was originally designed for tactile screens. Observations from screen recordings suggest imitating touch gestures by mouse is not intuitive, and itself a hindering factor.

6.2 Developer Perceptions and Bias

The participant feedback is widely consistent with screen-recording observations. However, some remarks readjusted our expectations and understanding.

Issues with the DSL Conversion Technique. Participants repeatedly noted difficulties navigating the tool's menus and reported accidental deletion of already modelled solutions, which was confirmed by the recordings. For some participants, this resulted in limited data loss. Overall, we consider that the main challenge using the DSL toolchain was its instability and handling of the tool's user interface. Despite this drawback, all participants succeeded in the requested migration.

Perceived Time Loss. As previously discussed, the major time losses observed and reported are problems with *Spring integration*, *IDE syntax highlighting* and using the *DSL tool's user interface*. Interestingly, a frequent mention of perceived time loss was the training material, notably the task illustration videos. More than two-thirds of the participants either skipped parts of the video, increased playback speed, or interleaved task solving with watching the video instructions. We will discuss in Section 7 how this affects the validity of our study.

Preference for Manual Conversion. We analyzed the final participant feedback regarding their preferences concerning both migration techniques. While the majority of participants deemed the DSL-driven approach easier (+25/1/-1) or more intuitive (+24/0/-3), the confidence towards a better performance of the DSL solution concerning our unit tests was balanced (+13/1/-13), and a majority of participants would not use the DSL technique in its current state for their own future projects (+6/6/-15). These statistics are illustrated in Figure 4. Note that the last question, concerning the participants' future project preferences was worded regarding the provided solutions. Participant feedback therefore might be influenced by negative experiences made due to the DSL tool's academic nature, rather than a purely methodological perception.

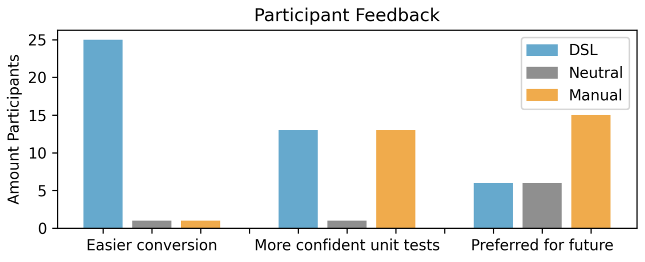


Figure 4: Participant Feedback on Individual Techniques

The participant's confidence in the correctness of their code significantly deviates from the measured correctness. Furthermore, we expected participants would also voice a preference for the migration technique they considered easier. However, this is not the case: The vast majority perceived the DSL toolchain technique as easier, but two-thirds stated they prefer to apply the manual methodology in the future. Further analysis of free text participant feedback suggests that developers associate the manual approach to feeling more in control:

Green Unicorn: "*(I prefer) the manual solution, because it gives more control over the source code.*"
Green Turtle: "*I will most likely stick with IntelliJ as I feel more comfortable coding everything manually where I have more control.*"
Yellow Turtle: "*Because the code generation process is unknown to me, I'd be more confident in the manual methodology [...], where I had total control and knew the code that would run against the tests.*"

Simultaneously, the main reason for mistrust in the DSL toolchain seemed to be the opacity of the model transformations, especially concerning code generation. In some cases, the feedback would even first acknowledge the advantages of the DSL approach, and then state a preference for manual migration:

Yellow Fox: "*I'm always suspicious of auto-generated code*"
Yellow Zebra: "*(I'm more likely to apply) IntelliJ because it feels more natural [...].*"
Green Zebra: "*TouchCORE (DSL technique) is more intuitive as it is more visual, but IntelliJ (manual) approach helps better to understand the underlying mechanism.*" and: "*(With the manual approach) I know clearer what is going on behind the scenes compared with the second (DSL) approach.*"
Blue Zebra: "*There's a lot of boilerplate code in restifying a legacy application, TouchCORE (the DLS tool) makes this easier and less error-prone.*" and: "*(I'm more likely to apply) manual, I've had problems with code generation tools in the past*"

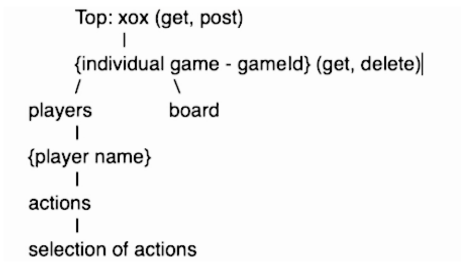
In summary, the developers majorly consider the DSL approach easier, but at the same time express a preference for the manual counterpart. The main reasons are 1) the association of coding with "being in control" combined with overconfidence in the own coding skills, and 2) a general mistrust in opaque transformations, especially in code generators.

6.3 Towards Improved Toolchain Acceptance

We see the previous discussion insights as an important finger-post towards further improvement of DSL toolchain approaches. The user feedback suggests the practical acceptance of DSL toolchains is not just a matter of performance benefits, but also apparel and applicability. These findings are coherent with previous findings, concerning the general acceptance of MDE tools in industry [33, 45, 46]. Based on the full feedback we received, and detailed analysis of our screen recording observation, we infer three main practical principles to obtain user acceptance, specifically for the design of future DSL toolchains:

Emphasize traceability and transparency. : We believe any toolchain must be as transparent as possible to the end user. Developers trust the compiler, partially due to the obvious link from code statements to execution instructions. Using a debugger, that link is so transparent, that users even forget the existence of underlying transformations. In comparison, the model transformation and code generation taking place in the presented DSL toolchain are opaque and thus hard to follow by the user. *To win user acceptance, a DSL toolchain must provide clear, immediate traceability of all modelling choices down to the generated code, to re-establish the developer's perception of being in control.*

Don't disrupt, integrate: Developers consider their preferred IDE a trusted environment, and are naturally reluctant to abandon this comfort zone. *Rather than proposing an orthogonal approach and fully replacing the IDE, a DSL toolchain should integrate as much as possible with the existing tools.* Multiple participants voiced they were more likely to embrace the benefits of the DSL-based approach if the toolchain were integrated as an IDE plugin, rather than a standalone software. This option combines well with the previous point, as a plugin could transparently highlight the impact of any modelling on the corresponding generated code, preserving full control over the sources. Interestingly, we even collected anecdotal evidence for the purposefulness of the ResTL DSL for manual migration - one participant opened a text editor in their second, manual migration task, to create an ASCII imitation of the structural ResTL DSL, as an intermediate migration activity. Figure 5 shows a capture of their recorded activity.



```
        Top: xox (get, post)
                |
        {individual game - gameId} (get, delete)
        /               \
players             board
    |
{player name}
    |
actions
    |
selection of actions
```

Figure 5: Participant Imitating the ResTL DSL in a Text Editor

Trust through UX:. Participant feedback predominantly judged the DSL and mapping process as highly intuitive. However, the user interface and stability of the toolchain were criticized. Especially the crashes and the gesture-based user interface were perceived as tedious respectively unintuitive. While this did not diminish the outcome of the DSL-based conversion process, we believe that the instability of the tool influenced trust in the generated outcome. *DSL toolchains cannot credibly claim to foster quality software engineering unless their implementation is held to the same standards.* Especially in comparison to established, industrial-grade IDEs, a DSL toolchain approach can only gain acceptance if stability and usability are equally excellent.

In summary, we believe the mistrust is not due to the DSL and the associated modelling and model transformation mechanisms themselves, but due to the way they are presented. Developers consider IDEs as their trusted work mode. Therefore, the most straightforward way to improve on acceptance would be to focus on IDE plugins that include mechanisms to transparently visualize the resulting transformations at the code level. Such a tool could offset the factors that we associate with the observed developer bias and possibly leverage dormant modelling toolchain potential.

7 Threats to Validity

We structure the discussion of factors that potentially weaken the validity of our findings, according to the four common types of validity, i.e., construct validity, internal validity, conclusion validity and external validity [49].

Construct Validity. Duration of a software task and test pass-ratio are both established and common constructs to measure the success and efficiency of a software development task [8, 21]. We are only aware of one participant, who did not try to follow the instructions in a meaningful way. Analysis of screen recordings and inspecting of the model and code submission suggested the participant rushed trough preparation and tasks, and seemingly had no interest in following the instructions. We considered the corresponding data and outlier and excluded it from analysis and interpretation.

Internal Validity. The varying skill levels of developers could influence individual performance. Pearson tests reported no significant correlation between any of the self-reported skill levels and the measured time or pass rate.

In one aspect the DSL toolchain behaved differently on Windows. Participants using Windows did not see method parameter names when mapping from ResTL to legacy functionality, they only saw enumerated stub names and parameter type. However, the omited information was easily accessible in the task material. Screen recording footage suggests the slowdown for Windows participants is negligible, for they immediately consulted the provided additional software documentation.

The majority of participants did not consume the video instructions strictly as intended. Increasing playback speed was common, as well as skipping introductory parts of the video instructions. In rare cases we observed participants interleaving the video instructions with solving their own task. We were able to restore the effective time spent on the migration task by details analysis of the screen recordings. It is hard to quantify to which extent this phenomenon influenced our findings. Additional Pearson tests (provided in the replication package) did not report a significant correlation between task familiarization and task-solving time or pass rate, i.e. we cannot conclude that participants are significantly slower in their task-solving when they consumed the task training more rapidly. We observed though that sloppy task familiarization had anecdotal effects on both techniques, e.g. having to deal with technical problems whose solutions were clearly explained in the training.

The task instructions should not favour one of the conversion techniques. We therefore set on a textual description of the migration target API outcome (see Section 4.3).

We acknowledge an unintentional difference in manual migration complexity for the two objects because Xox did not exhibit the dependency injection challenge. We believe this is a relevant factor for the comparatively low differences in average migration time differences for the Xox application. Concerning all other metrics, e.g. magnitude of interface size or code base complexity, the Bookstore and Xox are highly comparable.

Conclusion Validity. The statistical analysis was conducted with great care. We followed the best practices for the given crossover layout and additionally applied statistical inference corrections within test families. However, overly conservative corrections also bear a risk of introducing type-II errors (failing to reject a hypothesis although it is not true). An example is the first GLM model, which we did not interpret although the model characteristics would have supported our initial hypothesis and would have been consistent with the remaining analysis. We considered only significant results.

External Validity. The largest threat to the external validity, that is, the generalization of our experiment results to other contexts, is the recruited population. Skill distributions of students and professions are known to differ, and the use of experiments only with students is often debated regarding external validity [10]. We mitigated this risk with efforts to recruit diverse developer profiles, i.e., we were sending out invitations to developers from various backgrounds. Some factors are beyond our control, e.g. industrial engineers might feel less attracted by the compensation offered.

8 Conclusion

We presented a controlled experiment with 28 software developers to better understand DSL toolchain effects. We investigated the initial hypothesis of improved developer performance when a representative software engineering task is addressed with a corresponding DSL toolchain, rather than using manual code changes.

In our case, statistical analysis of developer performance shows a significant beneficial effect of the DSL toolchain, regarding the test pass-rate of the outcome. Concerning the time needed for migration to REST, we likewise observe a significant positive effect of the DSL toolchain for one sample application, and consistently lower variability of the process duration for both sample applications.

We discussed the performance differences by relating to qualitative observations, extracted from recorded participant task activity. We identified several plausible effects of toolchain components, notably the DSL obsoleting error-prone annotation syntax constructs and automated code generation of critical configuration files. We see the measurable effects and their associated causes as relevant empirical evidence for generally assumed DSL and MDE merits. Subsequently, we compared measured performance offsets to participant feedback and concluded a comparatively low acceptance for the DSL approach. We identified developer confidence in code, and general mistrust of MDE transformations, especially code generators, as the main factors for preferring the manual approach. We then derived recommendations for developers of DSL-associated tools. Finally, we presented the threads to the validity.

In the future, we could imagine accompanying our work with a longitudinal study, where we observe how a static population evolves in performance and perception over a prolonged time. Also, we would like to extend our observations to further DSL toolchains and study the influence of demographic factors by extending and further diversifying the population.

Acknowledgments

The authors were partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grant program. Jörg Kienzle's work was also partially funded by Junta de Andalucía, Spain, under project QUAL21 010UMA, and by the University of Málaga.

References

[1] Colin Atkinson and Thomas Kühne. 2008. Reducing accidental complexity in domain models. *Software & Systems Modeling* 7, 3 (2008), 345–359. https://doi.org/10.1007/s10270-007-0061-0

[2] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélène Raynal. 2015. MDE in Practice for Computational Science. *Procedia Computer Science* 51 (2015), 660–669. https://doi.org/10.1016/j.procs.2015.05.182 International Conference On Computational Science, ICCS 2015.

[3] Antonio Bucchiarone, Kemal Soysal, and Claudio Guidi. 2020. A Model-Driven Approach Towards Automatic Migration to Microservices. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer (Eds.). Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: Second International Workshop, DEVOPS 2019, Vol. 12055. Springer International Publishing, Cham, 15–36. https://doi.org/10.1007/978-3-030-39306-9_2 Series Title: Lecture Notes in Computer Science.

[4] Cristina Cachero, Santiago Meliá, and Jesús M. Hermida. 2019. Impact of model notations on the productivity of domain modelling: An empirical study. *Information and Software Technology* 108 (April 2019), 78–87. https://doi.org/10.1016/j.infsof.2018.12.005

[5] Donald T. Campbell and Julian C. Stanley. 2011. *Experimental and quasi-experimental designs for research.* Wadsworth, Belomt, CA.

[6] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2013. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering* (Feb. 2013). https://doi.org/10.1007/s10664-013-9248-x

[7] Thomas Cook, Donald Campbell, and William Shadish. 2001. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Wadsworth Publishing.

[8] África Domingo, Jorge Echeverría, Óscar Pastor, and Carlos Cetina. 2021. Comparing UML-Based and DSL-Based Modeling from Subjective and Objective Perspectives. In *Advanced Information Systems Engineering*, Marcello La Rosa, Shazia Sadiq, and Ernest Teniente (Eds.). Springer International Publishing, Cham, 483–498.

[9] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

[10] Robert Feldt, Thomas Zimmermann, Gunnar R. Bergersen, Davide Falessi, Andreas Jedlitschka, Natalia Juristo, Jürgen Münch, Markku Oivo, Per Runeson, Martin Shepperd, Dag I. K. Sjøberg, and Burak Turhan. 2018. Four commentaries on the use of students and professionals in empirical software engineering experiments. *Empirical Software Engineering* 23, 6 (Dec. 2018), 3801–3820. https://doi.org/10.1007/s10664-018-9655-0

[11] Roy Thomas Fielding. 2000. *Architectural styles and the design of network - based software architectures.* Ph.D. University of California, Irvine, United States – California. https://www.proquest.com/docview/304591392/abstract/D42F954B21D64B61PQ/1 ISBN: 9780599871182.

[12] Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. 2017. Reflections on the REST architectural style and "principled design of the modern web architecture" (impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017).* Association for Computing Machinery, New York, NY, USA, 4–14. https://doi.org/10.1145/3106237.3121282

[13] Marios Fokaefs and Eleni Stroulia. 2015. Using WADL specifications to develop and maintain REST client applications. In *2015 IEEE International Conference on Web Services.* IEEE, Piscataway, NJ, USA, 81–88.

[14] M. Fowler. 2010. *Domain-Specific Languages.* Pearson Education. https://books.google.ca/books?id=ri1muolw_YwC

[15] Mahdi Fahmideh Gholami, Farhad Daneshgar, Ghassan Beydoun, and Fethi Rabhi. 2017. Challenges in migrating legacy software systems to the cloud — an empirical study. *Information Systems* 67 (July 2017), 100–113. https://doi.org/10.1016/j.is.2017.03.008

[16] Jeff Gray. 2007. Domain-Specific Modeling. *Handbook of dynamic system modeling* 7, Handbook of dynamic system modeling (2007).

http://scholar.googleusercontent.com/scholar?q=cache:Ss4EaIQUb80J:scholar.google.com/+Jeff+Gray,+Juha-Pekka+Tolvanen,+Steven+Kelly,+Aniruddha+Gokhale,+Sandeep+Neema,+and+Jonathan+Sprinkle.+2007.+Domain-Specific+Modeling&hl=en&as_sdt=0,5

[17] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. 2008. DSLs: The Good, the Bad, and the Ugly. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA Companion '08)*. Association for Computing Machinery, New York, NY, USA, 791–794. https://doi.org/10.1145/1449814.1449863

[18] John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming* 89 (Sept. 2014), 144–161. https://doi.org/10.1016/j.scico.2013.03.017

[19] John Hutchinson, Jon Wittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical Assessment of MDE in Industry *(33rd International Conference on Software Engineering (ICSE 2011))*. IEEE, Piscataway, NJ.

[20] A. Jedlitschka and D. Pfahl. 2005. Reporting guidelines for controlled experiments in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, Noosa Heads, QLD, Australia, 10 pp. https://doi.org/10.1109/ISESE.2005.1541818

[21] Stephen H. Kan. 2003. *Metrics and Models in Software Quality Engineering* (second edition ed.). Addison Wesley.

[22] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton. 1996. A Software Engineering Experiment in Software Component Generation. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany) *(ICSE '96)*. IEEE Computer Society, USA, 542–552.

[23] Jörg Kienzle. 2023. TouchCORE. http://touchcore.cs.mcgill.ca/

[24] Jörg Kienzle, Gunter Mussbacher, Benoit Combemale, and Julien Deantoni. 2019. A unifying framework for homogeneous model composition. *Software & Systems Modeling* 18, 5 (03 Jan 2019), 3005–3023. https://doi.org/10.1007/s10270-018-00707-8

[25] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empir. Softw. Eng.* 20, 1 (2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3

[26] Tomaž Kosar, Sašo Gaberc, Jeffrey C. Carver, and Marjan Mernik. 2018. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering* 23, 5 (2018), 2734–2763. https://doi.org/10.1007/s10664-017-9593-2

[27] Tomaž Kosar, Marjan Mernik, and Jeffrey C. Carver. 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering* 17, 3 (June 2012), 276–304. https://doi.org/10.1007/s10664-011-9172-x

[28] Simon Maple and Andrew Binstock. 2018. JVM Ecosystem Report 2018 - About your Platform and Application. https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application/ https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application/.

[29] Yulkeidi Martínez, Cristina Cachero, and Santiago Meliá. 2014. Empirical study on the maintainability of Web applications: Model-driven Engineering vs Code-centric. *Empirical Software Engineering* 19, 6 (Dec. 2014), 1887–1920. https://doi.org/10.1007/s10664-013-9269-5

[30] Mark Masse. 2011. *REST API Design Rulebook.* O'Reilly Media. https://www.oreilly.com/library/view/rest-api-design/9781449317904/

[31] Andreas Menychtas, Christina Santzaridou, George Kousiouris, Theodora Varvarigou, Leire Orue-Echevarria, Juncal Alonso, Jesus Gorronogoitia, Hugo Bruneliere, Oliver Strauss, Tatiana Senkova, Bram Pellens, and Peter Stuer. 2013. ARTIST Methodology and Framework: A Novel Approach for the Migration of Legacy Software on the Cloud. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing.* IEEE, Timisoara, Romania, 424–431. https://doi.org/10.1109/SYNASC.2013.62

[32] Mulesoft. 2023. RAML Language Definition. https://github.com/raml-org/raml-spec

[33] Richard F. Paige and Dániel Varró. 2012. Lessons learned from building model-driven development tools. *Software & Systems Modeling* 11, 4 (Oct. 2012), 527–539. https://doi.org/10.1007/s10270-012-0257-9

[34] API Platform Postman. 2023. 2023 State of the API Report | API Technologies. https://www.postman.com/state-of-api/api-technologies/ https://www.postman.com/state-of-api/api-technologies/.

[35] Kate Revoredo, Djordje Djurica, and Jan Mendling. 2021. A study into the practice of reporting software engineering experiments. *Empirical Software Engineering* 26, 6 (Nov. 2021), 113. https://doi.org/10.1007/s10664-021-10007-3

[36] Roberto Rodríguez-Echeverría, Fernando Maclas, Vlctor M. Pavón, José M. Conejero, and Fernando Sánchez-Figueroa. 2014. Generating a REST Service Layer from a Legacy System. In *Information System Development*, María José Escalona, Gustavo Aragón, Henry Linger, Michael Lang, Chris Barry, and Christoph Schneider (Eds.). Springer International Publishing, Cham, 433–444. https://doi.org/10.1007/978-3-319-07215-9_35

[37] Maximilian Schiedermeier. 2024. Artifacts for RESTify Experiment. Data, custom tools, and material used and collected throughout the experiment. https://doi.org/10.5281/zenodo.12555385 Publisher: Zenodo.

[38] Maximilian Schiedermeier, Jörg Kienzle, and Bettina Kemme. 2021. FIDDLR: streamlining reuse with concern-specific modelling languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 164–176. https://doi.org/10.1145/3486608.3486913

[39] S M Sohan, Frank Maurer, Craig Anslow, and Martin P. Robillard. 2017. A study of the effectiveness of usage examples in REST API documentation. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, 53–61. https://doi.org/10.1109/VLHCC.2017.8103450

[40] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of Software Engineering Research. *ACM Transactions on Software Engineering and Methodology* 27, 3 (July 2018), 1–51. https://doi.org/10.1145/3241743

[41] Swagger. 2023. API Code & Client Generator | Swagger Codegen. https://swagger.io/tools/swagger-codegen/ https://swagger.io/tools/swagger-codegen/.

[42] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.* 35, 6 (jun 2000), 26–36. https://doi.org/10.1145/352029.352035

[43] Sira Vegas, Cecilia Apa, and Natalia Juristo. 2016. Crossover Designs in Software Engineering Experiments: Benefits and Perils. *IEEE Transactions on Software Engineering* 42, 2 (Feb. 2016), 120–135. https://doi.org/10.1109/TSE.2015.2467378 Conference Name: IEEE Transactions on Software Engineering.

[44] Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, Vedran Pavić, Jay Bryant, and Madhura Bhave. 2018. Spring Boot Reference Guide. (2018).

[45] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. 2013. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In *Model-Driven Engineering Languages and Systems*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke (Eds.). Vol. 8107. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-41533-3_1 Series Title: Lecture Notes in Computer Science.

[46] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. 2017. A taxonomy of tool-related issues affecting the adoption of model-driven engineering. *Software & Systems Modeling* 16, 2 (May 2017), 313–331. https://doi.org/10.1007/s10270-015-0487-8

[47] D. Wile. 2003. Lessons learned from real DSL experiments. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the.* 10 pp.–. https://doi.org/10.1109/HICSS.2003.1174893

[48] Claes Wohlin, Martin Höst, and Kennet Henningsson. 2003. Empirical Research Methods in Software Engineering. In *Empirical Methods and Studies in Software Engineering*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Reidar Conradi, and Alf Inge Wang (Eds.). Vol. 2765. Springer Berlin Heidelberg, Berlin, Heidelberg, 7–23. https://doi.org/10.1007/978-3-540-45143-3_2 Series Title: Lecture Notes in Computer Science.

[49] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering.* Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-29044-2