Development and evolution of Xtext-based DSLs on GitHub: an empirical investigation

Citation for the original published paper (version of record):
Zhang, W., Strüber, D., Hebig, R. (2026). Development and evolution of Xtext-based DSLs on GitHub: an empirical investigation. Empirical Software Engineering, 31(3).
http://dx.doi.org/10.1007/s10664-025-10775-2

Empirical Software Engineering (2026) 31:48
https://doi.org/10.1007/s10664-025-10775-2

Development and evolution of Xtext-based DSLs on GitHub: an empirical investigation

Weixing Zhang[1] · **Daniel Strüber**[1,2] · **Regina Hebig**[3]

Received: 28 January 2025 / Accepted: 11 November 2025

Abstract

Domain-specific languages (DSLs) play a crucial role in facilitating a wide range of software development activities in the context of model-driven engineering (MDE). However, there exists a significant gap in the systematic understanding of how DSLs evolve over time, which could hamper the development of effective methodologies and tools. To address this gap, this paper presents a large-scale study of the development and evolution of textual DSLs created with the Xtext framework and hosted on GitHub. The study focuses on how these languages evolve at the grammar and front-end level, as captured in open-source repositories. We systematically identified and analyzed 1002 GitHub repositories containing Xtext-related projects. A manual classification of the repositories brought forward 226 ones that contain a fully developed language. We further categorized the latter into 18 separate categories of application domains, studied their contained DSL definition artifacts and analyzed the extent to which example instances using the grammar are available. In addition, we explored DSL development practices, focusing on the development scenarios involved, evolution activities, and the modification and co-evolution of related artifacts. We observed that analyzed DSLs evolved faster and were maintained longer when they belonged to specific domains, such as data management and databases. We found grammar definitions of DSLs in 722 repositories in total. While only about a third of them provided corresponding textual instances, community engagement metrics indicate potential usage of the DSLs in downstream repositories. Considering different language development approaches, we found that the majority of analyzed languages were developed following a grammar-driven approach, although a notable number adopted a metamodel-driven approach. Additionally, we identify a trend of retrofitting existing languages in Xtext, illustrating the framework's flexibility beyond the creation of new DSLs. By investigating software evolution aspects, we found that the development lifecycle of analyzed DSLs varies, but in many cases, updates to grammar definitions and example instances were frequent, and most of the evolution activities can be classified as "perfective" changes. Addressing a need for large and systematically documented datasets in the model-driven engineerifng community, we contribute a dataset of repositories together with our collected meta-information, which can be used to inform our understanding of

Communicated by: Alexander Serebrenik.

Extended author information available on the last page of the article

Published online: 16 December 2025

open-source DSL development practices and the development of improved tools for supporting the development and evolution of DSLs.

Keywords Xtext · Software evolution · DSLs

1 Introduction

Domain-specific languages (DSLs, Kosar et al. 2016) are custom-tailored software languages addressing a particular domain of expertise. By providing a tool to create models on a suitable abstraction level, DSLs play an important role in model-driven engineering (MDE, Stahl and Völter 2006), where models created using a DSL can be used for a large variety of activities such as design, analysis, code generation, and testing.

Developing a DSL is a high-stakes activity. Previous design decisions often cannot be changed without significant effort on the part of the language developers and users. Still, a need to change the language may arise especially in the context of *language evolution*, where the developers add new features or respond to experience with the language (Lämmel 2018). In consequence, there is a need for sound methods, practices, and techniques for supporting the evolution of DSLs. However, to date, the development of support for DSL evolution is typically driven by the opinion of experts and individual cases encountered in their own practice or experience reports—Thanhofer-Pilisch et al. (2017) provide a survey with 14 individual cases. Developers of future evolution methods would benefit from systematic knowledge about DSL evolution obtained from a larger number of cases.

To understand how MDE artifacts are developed and evolved, there is a trend towards large-scale studies that systematically collect evidence from open-source software (OSS) projects (Hebig et al. 2016b; Shrestha et al. 2023; Babur et al. 2024). However, for development of DSLs in the context of MDE, such a study is not available yet. In the context of DSL evolution, as we will explore in this paper, such a study can identify patterns and frequencies that cannot be observed in small-scale qualitative studies.

In this paper, addressing this gap, we contribute the first large-scale multiple-case study of DSL development and evolution, focused on textual DSLs developed with the Xtext framework (Bettini 2016) hosted on GitHub. Xtext is widely used in the MDE community due to its strong integration with Eclipse and its ability to automatically generate substantial parts of the parser and front-end tooling (e.g., syntax highlighting, editor support, validation). This approach has also inspired more recent DSL workbenches such as *textX* (https://pypi.org/project/textX/) and *langium* (https://langium.org/). Consequently, our analysis concentrates on the development and evolution of DSL definitions (grammars and related artifacts, such as meta-models) as they are maintained in repositories, rather than on the evolution of generated tooling or additional back-end components that may be developed alongside.

Based on a repository mining methodology (Kalliamvakou et al. 2014), we collected and analysed data from 1002 GitHub repositories, spanning a large variety of application domains, from widespread ones such as database query languages to niche areas such as mod development for a particular computer game. We observed that GitHub-hosted Xtext DSLs span a spectrum from small experimental prototypes to long-lived, established languages

maintained by multi-person teams, e.g., `Applause`, `GraphQL-Xtext`, and `Epsilon` being examples for the latter. Therefore, our analysis focuses on different subsets of repositories depending on the research question: For artifact analysis, we consider 722 repositories containing Xtext files; for development scenario and evolution studies, we consider 226 repositories with a fully-developed language meeting a set of defined quality criteria.

As part of our contribution, we provide a dataset (Zhang et al. 2025) of 1002 repositories (via their URLs), including 226 repositories with fully developed languages, together with our extracted meta-data, e.g., the repository's type, employed development scenario, availability of various artifacts, and change statistics. This dataset addresses the need for large and consistently documented artifacts expressed in the MDE community (Robles et al. 2023; Damasceno and Strüber 2021) and can be particularly useful for follow-up research, both to develop advanced (e.g., AI-based) techniques, as well as supporting the identification of cases that can be used to inform design and evaluation activities.

To clarify our research scope, we distinguish between DSL development and DSL evolution following established software engineering definitions. Development refers to the initial creation and implementation of a language, including grammar design, tooling setup, and example creation (Völter et al. 2013). Evolution, in contrast, refers to the ongoing modification and enhancement of existing languages that typically evolve quite radically throughout their lifetime as they grow from small, declarative languages to acquire new features in response to users finding them useful and desiring more power (Tratt 2008).

With these definitions in mind, we focus on the following research questions:

RQ1 *Are there GitHub projects that use Xtext? Which are these projects?* We set out to investigate basic information on Xtext-related activities in GitHub repositories, including the type and application domains of these repositories and their contained languages. While our study focuses on Xtext-based DSLs, the repository classification and domain analysis approaches are designed to be framework-agnostic. This research question is broken down into two further research questions.

RQ1.1 *How can these repositories be categorized?* This classification will allow us to know which repositories contain well-documented languages, which repositories have developed infrastructure for Xtext-based DSLs, etc. This will be particularly useful for follow-up research (e.g., our RQ2 to 4).

RQ1.2 *Which trends define the application domains of these DSLs in recent years?* Answering this question can reveal in which domains Xtext-based DSLs are mainly developed, and in which domains DSL development is increasing, and whether it is increasing all the time. For example, data management and databases are increasingly important domains for Xtext-based DSLs. The frequency of language development is related to the evolution of the language. Answering this question can prepare us for studying the evolution of Xtext-based DSLs.

RQ2 *What language artifacts for Xtext-based DSLs do these repositories contain?* Our file extension-based artifact identification approach can be adapted to analyze DSL repositories in other frameworks, though the specific extensions would differ.

RQ2.1 *What main language definition artifacts do these repositories contain?* In answering this question, we investigate three key language artifacts of Xtext-based DSLs, namely, grammars, metamodels, and workflow definitions. This allows us to overview of the contained language artifacts in these repositories that are related to Xtext activities, and the answer to this question is particularly useful for follow-up research (e.g., our RQ3 and 4).

RQ2.2 *Do these repositories contain both grammars and instances that adhere to it?* The answer to this question can reveal which repositories contain both grammar and example instances and whether they contain both. Providing example instances is beneficial for using the DSL, so it is interesting to know whether the repositories already provide example instances that adhere to the grammar.

RQ2.3 *To what extent is Xtext grammar used by example instances?* When answering this question, we will know how many grammar rules in the grammar are used by the instance files in the same repository, i.e., the coverage of the instance to the grammar. High coverage of the grammar by example instances can indicate comprehensive illustration of the language's features within the repository, addressing a literature recommendation to provide documentation in the form of examples (Völter 2009) and empirical findings according to which novice DSL adopters use available examples as a starting point for developing their own programs (Rennels and Chasins 2023). Co-location in the same repository makes documentation easier to find (Paik and Wallin 2020), and helps mitigate the well-known problem of outdated examples reported in related contexts such as API usage (Radevski et al. 2016) by enabling practices where examples are treated as executable tests (e.g., Rust documentation tests Rust Project Developers 2024).

RQ3 *Which development scenarios for Xtext-based languages are applied in these projects?* Xtext supports multiple development scenarios that differ in their complexity (discussed later). There is a question on whether complex development scenarios such as metamodel-driven development are used in practice and thus, need to be supported with dedicated approaches. Moreover, in the course of answering this question, we discovered a trend of what we call *retrofitting*—creating an Xtext grammar that fits an existing language. The retrofitting phenomenon appears across different DSL frameworks where developers implement existing languages in new tooling environments.

RQ4 *How do Xtext-based languages on GitHub evolve over time?* Since DSLs are often envisioned as "small" languages (Deursen and Klint 1998), it is tempting to view their evolution as a non-issue. In this RQ, we study longitudinal aspects of language projects, including their longevity and amount of changes performed. We highlight the existence of long-living language projects and shed light on their proneness to significant changes, leading to challenges that we discuss later, in Section 6 of the paper.

RQ4.1 *How can grammar evolution in these projects be characterized quantitatively?* The answer to this question depicts the active time span of the grammar in these repositories, i.e., the time of first creation and the end time of the last evolution (with the commit time as reference). In addition, the answer to this question also reveals the quantity of grammar

evolution in these repositories, i.e., the number of changes in grammar rules and the number of changed lines of grammar definitions.

RQ4.2 *How common are different types of changes during the evolution of grammars?* An important aspect of evolution is the reason for changes. Software systems change because of changing requirements and user needs. However, they also change for maintenance reasons, e.g. when adaptations are necessary due to changes in used libraries or operating systems or when failures occur that need to be corrected. We ask the question whether evolution of DSLs is mostly due to maintenance reasons or whether changes to extend the languages are common as well.

RQ4.3 *How do textual instances co-evolve with grammar in real projects?* Xtext-based DSLs are developed in repositories and, like general-purpose languages, inevitably evolve over time. By focusing on repositories that contain DSLs, this question examines how their instances co-evolve with the grammar. We deliberately limit our scope to co-evolution within the same repository, excluding downstream projects, to provide a clear and controlled analysis. This focus reflects the central role of instances accompanying the language, as established in RQ2.3. Our goal is to establish whether these instances remain consistently aligned with the evolving grammar.

This paper is a significantly extended version of our previous conference paper (Zhang and Strüber 2024), in which we first presented our dataset and addressed a subset the research questions stated above. For the present manuscript, we considerably extended our analysis to provide a more in-depth look into the included projects, their included artifacts, and evolution activities. This entailed extensive work on classifying domains and analyzing their trends (leading to the new RQ1.2), retrieving language instances and analyzing their coverage of DSL concepts (RQ2.2 and RQ2.3, both new), analyzing community engagement patterns for repositories without co-located instances, and manual labeling of commits and studying files contained in them (RQ4.2 and RQ4.3, both new). Based on the new findings, we also significantly extended our discussion, to discuss the requirements for better supporting artifact co-evolution (significantly extended Section 6.1) and the need for a better understanding open-source of DSL development practices and ecosystem dynamics in practice (new Section 6.2) that inform tool development and research directions.

The rest of this paper is structured as follows. In Section 2, we introduce necessary background. In Section 3, we discuss related work. In Sections 4 and 5, we present the methodology and results for our study. In Section 6, we discuss implications of our results as well as threats to validity. In Section 7, we conclude.

2 Background

2.1 Xtext

Having established our rationale for focusing on Xtext, we now provide the necessary technical background. Xtext (Bettini 2016) is a framework for developing textual domain-specific languages. Xtext allows the specification of a textual DSL in terms of an extended

EBNF grammar, where the extensions are mappings of language elements to an underlying metamodel (in EMF Steinberg et al. 2008). The metamodel specifies the language's abstract syntax (language concepts and their relations), whereas the grammar specifies the concrete syntax (keywords, parentheses, nesting of elements) with the mapping to the abstract syntax. From the provided specification, Xtext can automatically generate comprehensive tool support, including a textual editor with automated checks, syntax highlighting, and autoformatting. While Xtext is rooted in the Eclipse ecosystem, adapters for other IDEs (e.g., IntelliJ) are available.

In our repository mining context, we identify grammars and metamodels, as well as two additional artifact types, as distinct file type, illustrated in Fig. 1. In Xtext-based DSL development, there are the following four main MDE artifacts: 1) Xtext files, 2) Ecore metamodel files, 3) modeling workflow engine (MWE) files, and 4) textual instance files. MWE files support the orchestration of automated activities, in particular, the generation of modeling components. Among others, they define the file extension for textual instances (a.k.a. models created using the language), which render them interesting for our study of artifacts. Xtext files (i.e., grammar file) can be generated from Ecore files, and conversely, Ecore files can also be generated from Xtext files (Bettini 2016). An MWE file is used to generate Xtext artifacts from Xtext grammar (Bettini 2016), and these Xtext artifacts are used to generate a textual editor for the DSL. Textual instance files are edited in this textual editor.

2.2 Development Workflows and Scenarios

To specify a DSL's concrete and abstract syntax, Xtext uses two separate artifacts–a grammar and metamodels—, which leads to the challenge of keeping them synchronized with each other as the language evolves. To this end, Xtext supports two main development workflows (Bettini 2016): In a *grammar-driven* scenario, the user primarily edits the grammar, and has changes propagated to the metamodel by completely re-generating it. In a *metamodel-driven* scenario, the user primarily edits the metamodel. After the changes to the metamodel, the grammar has to be updated, which in the default process has to be done manually–re-generating the grammar is not feasible due to potential information loss about the concrete syntax.

Choosing an appropriate workflow requires to consider the context in which the language is developed. The metamodel driven workflow is useful in scenarios where the metamodel has to be manually managed, e.g., when it is the center of an already-existing ecosystem of tools, when it comes from a third party vendor or standardization committee, or in a blended

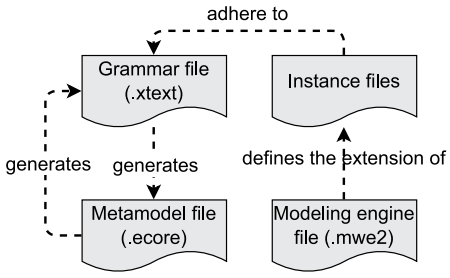

Fig. 1 Main MDE artifact types in Xtext-based projects

modeling scenario with several concrete syntaxes. The grammar-driven workflow is simpler to support and, therefore, generally preferable in other scenarios.

2.3 Software Maintenance Intentions

Following a categorization of Lientz and Swanson (1980) the intentions behind software maintenance and evolution can be classified into 4 groups:

Adaptive changes or maintenance describes changes that happened in reaction to a change in the software environment, such as a change in a used API. An example is from the "jkind-xtext"[1] repository, a `Lustre` plug-in for Eclipse. On March 19, 2016, a commit titled "Updates for JKind v3.0" specifically exemplifies an adaptive change. Developers updated the Bundle-Version in MANIFEST.MF from 2.3 to 3.0 and adjusted grammar files to accommodate the new JKind v3.0, ensuring the plug-in's continued compatibility and functionality.

Perfective changes or maintenance describes enhancements of a software system or reactions to changing requirements. An example comes from the "JSFLibraryGenerator"[2] repository, which hosts the JSFLibraryGenerator, an Xtext project designed to simplify JSF[3] component library creation. On March 5, 2016, a commit titled "added new types" to its grammar illustrates a perfective change. This modification involved adding a new data type to the DSL's grammar, extending its capabilities to generate more comprehensive JSF component artifacts.

Corrective changes or maintenance describes corrections of failures and errors of the system. For instance, in the "megal-xtext"[4] repository, a commit on January 24, 2017, specifically addressed a "megal import bug" within its megal.xtext grammar file. This change replaced an unreliable HTTP-based model import with a stable platform-specific resource path, resolving issues related to unstable external dependencies and ensuring consistent parsing and validation of the DSL.

Preventive changes or maintenance describes changes to the software that are meant to ease future changes or prevent problems. An example is from the "eclipse-typescript-xtext"[5] repository, an Xtext-based TypeScript parser. On January 4, 2014, a commit titled "just added TODO and link to JS possible grammar inspiration" to Typescript.xtext illustrates this. The developer added comments with a to-do note and a link to an external resource. This act, though not a code change, proactively guides future development and reduces potential design challenges in completing the TypeScript grammar.

[1] https://github.com/loonwerks/jkind-xtext

[2] https://github.com/stephanrauh/JSFLibraryGenerator

[3] https://www.oracle.com/java/technologies/javaserverfaces.html

[4] https://github.com/avaranovich/megal-xtext

[5] https://github.com/vorburger/eclipse-typescript-xtext

3 Related Work

3.1 DSL Evolution

The evolution of DSLs has been studied so far with a focus on providing improved evolution support, and on reporting individual cases of evolving DSLs. Both are studied in a systematic mapping study of Thanhofer-Pilisch et al. (2017), the results of which can help researchers and practitioners working on DSL-based approaches obtain an overview of existing research on and open challenges of DSL evolution. We now discuss selected cases with a particular focus on industrial experiences. Mengerink et al. investigated the evolution of DSLs in a large industrial MDSE ecosystem (Mengerink et al. 2018), through which they summarized common evolution types and evaluated the automation capabilities of evolution in real-life scenarios. Schuts et al. reported in Schuts et al. (2021) their experience in evolving a Philips-owned DSL, with the goal of enabling the DSL to support a range of Philips systems. This is concrete experience from the industry regarding the evolution of DSL.

More recent industrial work has continued to underscore the evolution challenges that arise when deploying DSLs in practice. Denkers (2024) conducted extensive case studies developing and evolving DSLs with language workbenches in an industrial printing systems context, identifying that successful DSL evolution requires improvements in non-functional aspects such as portability, usability, and documentation to sustain long-term industrial adoption. Ratiu et al. (2021) reported on their five-year experience with DSL instantiation and maintenance across three different business domains at Siemens, revealing how the need to adapt DSLs to changing domain requirements and evolving business needs presents ongoing evolution challenges for industrial practitioners. Akesson et al. (2020) specifically addressed the challenge of managing an evolving DSL ecosystem in the defense domain, identifying key research questions including how to achieve modularity and reuse in DSL ecosystems and how to manage consistency between models and realizations during evolution, further highlighting the critical importance of systematic approaches to DSL evolution in industrial contexts.

DSL evolution generally leads to issues with keeping multiple involved artifacts synchronized with each other (Lämmel 2018). In the MDE community, a plethora of work on co-evolution problems has evolved, in particular metamodel-model co-evolution (Wachsmuth 2007; Herrmannsdoerfer et al. 2009; Hebig et al. 2016a), where changes to the metamodel make evolution of the associated models and transformations necessary. As we discuss later, our dataset and results could inform the development of new approaches in this area.

While existing DSL evolution studies provide valuable insights from individual cases and controlled environments, they are limited in scope. Most studies focus on a single DSL or a small set of languages in a specific industrial setting (Denkers 2024; Mengerink et al. 2018; Ratiu et al. 2021; Schuts et al. 2021). In contrast, our large-scale approach is able to systematically identify patterns across different domains and development environments, such as the most common commit types for a language, that cannot be observed through case-by-case analysis. This scale enables us, within our GitHub-hosted Xtext project corpus, to distinguish common practices from exceptions, laying the foundation for evidence-based tool development and method design for the broader DSL engineering community.

3.2 Mining

Due to data availability and volume, GitHub has become a primary data source for repository mining research (Gousios and Spinellis 2017), including those in model-driven engineering. In Shrestha et al. (2023), Shrestha et al. mined MATLAB/Simulink-related repositories and assembled a large corpus of Simulink projects, which includes model and project changes and allows redistribution. Mengerink et al. used software repository mining to create a large corpus of OCL constraints (Mengerink et al. 2019). Previous studies of EMF metamodels focused on collecting EMF models from Eclipse projects (Kögel and Tichy 2018), studying the use of metamodeling concepts in GitHub projects (Babur et al. 2024), and on deriving a high-quality dataset for machine learning (López et al. 2022). Hebig et al. (2016b) investigated the use of UML in OSS projects by systematically mining GitHub projects. No previous study focused on Xtext-based DSLs in GitHub repositories.

The only previous work that explicitly applied repository mining to Xtext grammars (among other MDE artifacts) is MAR, a search engine for models (López and Cuadrado 2022). MAR offers a by-example query mechanism for searching a database of 600K models retrieved from existing repositories. While their underlying dataset includes Xtext models from GitHub, it is not annotated with the metadata offered in our dataset (e.g., number of instances, development scenario, evolution statistics). Moreover, our research contribution and questions have a different scope, focused on characterizing Xtext-specific projects, with their development and evolution scenarios.

4 Methodology

We now describe our used repository mining methodology (Gousios and Spinellis 2017). We chose GitHub as our data source because it has become the dominant platform for open-source development and is widely used in repository mining research (Hebig et al. 2016b; Shrestha et al. 2023; Babur et al. 2024). For DSLs in particular, GitHub offers (i) a unique concentration of Xtext-based projects across a wide variety of domains and (ii) long-lived projects with sustained maintenance activity. This combination makes GitHub a meaningful and practical source for studying DSL development and evolution.

Our analysis is designed to address different research questions using appropriate repository subsets identified through our methodology. We use the complete set of initially collected repositories for classification analysis (RQ1.1), repositories containing Xtext files for artifact analysis (RQ2), and repositories classified as containing fully developed languages for domain trend analysis (RQ1.2), development scenarios (RQ3), and evolution studies (RQ4). Our overall process, shown in Fig. 2, is divided into 7 steps. First, we obtained a list of non-fork open-source repositories containing Xtext files from GitHub. Second, we cloned all the retrieved open-source repositories to a local hard drive to facilitate access and information acquisition. Third, we manually classified all obtained repositories, before analyzing the collected data with respect to our research questions. Fourth, we searched for relevant file types in these repositories by file extensions and collected information about them. Fifth, using the repository-specific file extensions of the instances that we identified from MWE files, we collected the instances of these extensions and calculated information

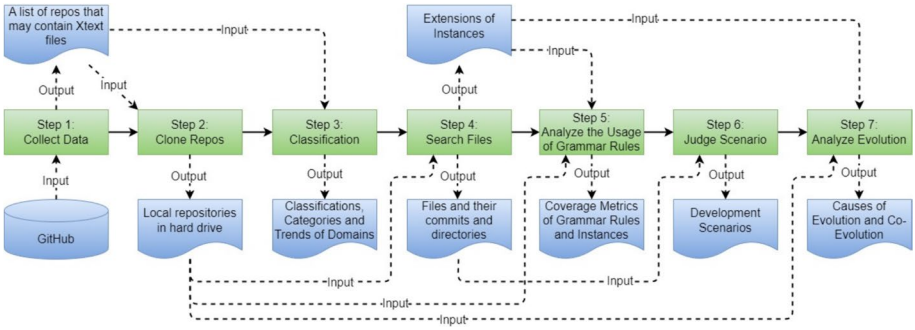

Fig. 2 Overall process

about them. Sixth, we analyze the development scenario of each repository. Seventh, we analyze the evolution of the languages.

4.1 Step 1: Data Collection

As an initial data collection effort, we used the GitHub API to obtain repositories that are related to Xtext. The obtained list formed the basis for answering all of our research questions in detail in the subsequent steps.

Since the most fundamental MDE artifact of an Xtext project is its grammar, which is stored as a file with the extension *.xtext*, our search string was based on the main clause `q?=.xtext` – that is, we searched repositories that contain a file with that extension. We later observed that this search string partially led to the identification of repositories that did not actually contain an Xtext file, but were still related to Xtext in a different way, as described below. Furthermore, to exclude repositories that are forks of other repositories, as these might mostly replicate the information from the original repositories and thus bias our results, we set the parameter “fork" to “false".

One complication was that the GitHub API only allows access up to 1,000 results, even when using the *pagination* feature. However, from a trial search on the GitHub website, we observed that the number of relevant repositories may exceed 1,000. Hence, added a third parameter to the request, which is the creation time of the repository. We use January 1, 2018, as the boundary to divide the request into two, i.e., requesting results before that date and from that date. We retrieved six pages with 576 repositories created before January 1, 2018, and five pages with 426 repositories created from this date, leading to a total of 1002 repositories.

We developed a Python script to complete the above work. Its functions included setting request parameters, sending requests, and dumping request results into local text files. Execution of this script only took a few seconds to complete. The rationale for developing a new script, instead of starting from an existing dataset (e.g., López and Cuadrado 2022) was that it allowed us to retrieve the most up-to-date information from GitHub and that it naturally integrated with our remaining analysis activities. The overall query used for the requests had the following form:

```
https://api.github.com/search/repositories?  q=.xtext+cr
eated:{since_date}..{stop_date}  +fork:false&page={page}
&per_page=100
```

The repository information we obtained contains various information about the repository, such as the repository's ID, name, whether it is private, owner, and html_url. We developed another script to extract the name, owner's login, and html_url of these repositories from the text files and store them in a table to facilitate subsequent data mining and analysis. The result was a table containing 1002 rows.

4.2 Step 2: Repository Cloning

To facilitate comprehensive analysis for our four research questions, we cloned all repositories. While GitHub allows to obtain information about GitHub repositories through the GitHub API, it has restrictions on the rate and frequency of access. Since we in subsequent analysis needed to frequently access different and large numbers of files, we decided to clone all repositories to a local hard drive for more efficient analysis. We only cloned the `master` branch (except for a few cases where `master` was empty, where we manually identified a different main branch instead), leaving an analysis of use of branching and pull request as future work. This decision was made because: (1) master branch represents the main development line with successfully integrated changes, (2) pull request commits are typically merged into master upon acceptance, and (3) focusing on master branch allows us to study completed evolution activities rather than experimental or abandoned development efforts.

4.3 Step 3: Classification

In step 3, we classified the repositories and domains of the DSLs. The results of this step were directly reflected in our obtained overview of repositories in RQ1.1 and RQ1.2, and furthermore supported our filtering efforts in RQ3 and RQ4, where we focused on repositories that contain a fully developed language.

4.3.1 Classification of Repositories

To give deeper insights into the different kinds of Xtext-related repositories on GitHub, in Step 3, we manually classified repositories into different types and analyzed the frequency of different types. Our process for this was as follows: First, one author manually labeled a sample of 200 repositories with improvised labels. The labels emerged from the observations that a few specialized categories were recurring between the repositories, with proper language projects (described below) being of most interest for our study. Second, in a discussion between the authors, the obtained labels were harmonized, by defining descriptions and explicit criteria for them. Third, we labeled the complete set of all repositories with the final set of labels. The final labeling was done by one author and checked by another author. Disagreements were resolved together.

The obtained list of types together with their descriptions and criteria was as follows:

Language: A repository with proper, documented Xtext-based language.

- *Criterion:* The README.md or "About" section describes it as (implementation of) a language or a software system that incorporates a clearly identifiable language, and the repository is not of clearly experimental or personal nature.
- *Notes:* More detailed criteria for experimental or personal nature are documented below, under the label *Experimental/Personal*. We also collected the language's domain. After noticing that several repositories provided a re-implementation of an existing language (a phenomenon we call *retrofitting*), we also noted whether this is a case for each repository. We did not include a criterion of whether the repository includes an Xtext grammar, which allowed us to identify edge cases that we report on later, in Section 5.2.1.

Training/Examples: A repository serving the training of Xtext users, usually in the form of an example, tutorial or both.

- *Criterion:* The project's README.md or "About" section describes it as an example, a tutorial, or a demonstration.
- *Note:* Using the word "example" as part of the name was not deemed as a useful criterion, as examples might be created for experimental purposes—see below.

Infrastructure: A repository with tooling for supporting development with Xtext.

- *Criterion:* The README.md or "About" section suggests that the project is about supporting tooling.

Experimental/Personal: A repository that does not fall in any of the above categories, but is still directly related to the language workbench Xtext.

- *Criteria:* Any of the following applies:

  1. The contained grammar is extremely small and basic.
  2. The contained grammar is taken from a standard example provided with Xtext.
  3. The README.md and "About" section are empty or give no context information on what the repository is about.
  4. The README.md and "About" section describes it as an assignment submission for a course, or as an example for debugging purposes.

Unrelated: A repository unrelated to the language workbench Xtext, except for naming.

- *Criterion:* The only connection to the language workbench is sharing (parts of) the name.

4.3.2 Domain Identification and Categories

In RQ1.2, we focused on 226 repositories classified as "Language". We manually reviewed these 226 repositories to identify their domains. To reduce the bias caused by manual identification, one of us performed the initial identification for each repository, while one of the other authors reviewed each identified domain. During the review process, when there was a disagreement, we discussed and voted on the final domain determination within the author team.

The resulting list of domains was at a fine-grained level, with entries such as *Security protocol analysis*, *SAT solving and verification*, and *Telemedicine System*. To overview the variety of domains and to analyze the trends in application domains over time, we needed to categorize these fine-grained domains into coarse-grained categories, such as *Security and Networking* and *Artificial Intelligence and Machine Learning*. To automatically derive suggestions for such categories, we adopted an LLM-based method proposed by Chen et al. (2023). This is a task where using an LLM is suitable because (i.) multiple valid solutions are permissible; (ii.) it is not an ad hoc solution, but supported by Chen et al.'s research that we built on; (iii.) manually reviewing multiple LLM-generated solutions and selecting a most feasible one among them is feasible, and has been performed by us in this study. The mapping of domains to the identified categories has been done manually as well, leading to an additional manual plausibility check. Specifically, the used LLM was ChatGPT-4.0. We fed our list of 226 fine-grained domains into a prompt of the form "*I have 226 domains, which are placed in the attached TXT file. Could you categorize these 226 domains into larger categories?"* We repeated this approach five times, manually reviewed the resulting lists and voted on them to determine the best suggestion. This resulted in the following categories: "Programming Languages", "Software Development and Engineering", "Games", "Web and Mobile Development", "Modeling, Simulation, and Design", "Data Management and Databases", "Security and Networking", "Artificial Intelligence and Machine Learning", "Business and Enterprise Applications", "Healthcare and Life Sciences", "IoT, Embedded Systems, and Hardware", "Mathematics, Logic, and Scientific Computing", "Testing and Verification", "Content, Information and Document Management", "Cloud, APIs, and Web Services", "Graphical User Interfaces (GUI)", "Questionnaire", and "Miscellaneous".

We then manually mapped the 226 domains to these 18 categories. This was again done first by one of us and followed up by a review of the mapping performed by another author. Again, if there was a disagreement, we decided on the final mapping by discussing and voting within the author team.

4.4 Step 4: File Search

To address RQ2.1 and RQ2.2, which focus on artifacts contained in Xtext-related projects, we systematically searched the cloned repositories for different types of files: main language definition artifacts (grammars, metamodels, and workflow files) and textual instances.

4.4.1 Xtext/Ecore/MWE2 File Search

Given the local clones of the 1002 identified repositories, in Step 4, we identified their contained grammars, metamodels, and workflow files, by searching for files with the extensions ".xtext", ".ecore", ".mwe2", respectively.

Moreover, we did additional analysis, specifically, whenever we found an Ecore metamodel file, in addition to recording the number of commits, we also recorded the name of the folder containing the file for subsequent analysis of the language development scenario; whenever we found an MWE2 file, in addition to recording the number of commits, and we also looked for the instance extension it defined.

4.4.2 Instance Search

One Xtext artifact type we set out to study was instances (models); yet, identifying instances is non-trivial, as their file extension differs per language. We identified instances by reading the file extension from the previously identified MWE files and then searching relevant files. To check whether the found instances adhere to the grammar in the same repository, we performed a sampling analysis: we randomly selected ten repositories that contained both grammar and instances and manually checked conformance. For all ten repositories we found that the contained instances fully conformed to the contained grammars. Thus we will make the assumption that instances files found in a repository with an Xtext grammar are most likely written in the language specified by that grammar.

Our instance identification approach is limited to artifacts co-located within the same repository as the DSL definition, in line with our motivating scenario, in which novice users inspect example instances made available with the DSL. This methodological constraint implies that our analysis cannot account for instances that may be maintained in separate downstream repositories, private enterprise projects, or distributed across different hosting platforms. Systematic mining of GitHub for such distributed instances could not guarantee completeness due to private repositories (including private forks of our considered public ones) and non-GitHub hosting. Therefore, we focus our analysis on co-located artifacts, which provides a reliable foundation for understanding DSL development patterns within individual repositories.

4.5 Step 5: Analyze the Usage of Grammar Rules

To answer RQ2.3, we need to obtain the total number of grammar rules in each grammar (i.e., the xtext file) and the number of types of objects in the instance (i.e., the grammar rules used). To this end, for those repositories categorized as "Languages", we first filter out those that contain both grammars and instances. We developed a script that counts the total number of grammar rules in all Xtext files in a single repository. When counting grammar rules in xtext files, we skipped the xtext files in the paths `src-gen` and `bin` because they are backups of the xtext files in `src`.

At the same time, for each grammar, we need to obtain each element in the instance that complies with it and the type of these elements. The types in the instance correspond one-to-one to the grammar rules in the grammar. Obtaining the total count of all types used in the instances and the total count of grammar rules in the grammar can be used to calculate

the coverage of the instance to the grammar. Our script can traverse all child elements in the found instance and obtain their types. Then place the types in a list by removing the duplicate ones. Ensuring that the project where the Xtext grammar is located has no errors and can be resolved normally is a prerequisite for obtaining the element types in the instance. A technical problem encountered was that the Xtext projects in the repository could not be resolved on the experimental machine (i.e., the author's computer) due to Xtext version issues. The Xtext version on the experimental machine is 2.36, while many repositories contain the Xtext projects that were created with an older version of Xtext. Given the practical difficulties of downloading and installing old versions of Xtext, we re-created these Xtext projects with Xtext 2.36 by fully reusing their grammars. Then we used our script to call the re-created Xtext projects to parse the instances in the original repository to calculate the number of used grammar rules.

Finally, for each repository, we calculate the coverage of grammar rules by textual instances. Specifically, for a repository containing grammars and instances, we first count the total number of grammar rules *R* across all grammars in that repository. Next, for each grammar, we identify all unique types used across all textual instances in the repository, obtaining the total count *T* of unique types. Since each type in the instances corresponds to a grammar rule, we calculate the percentage of grammar rules used as:

$$\text{Grammar Rule Coverage} = \frac{|\bigcup_{j=1}^{m} T_j|}{\sum_{i=1}^{k} R_i} \times 100\%$$

where R_i represents the number of grammar rules in the *i*-th grammar file, *k* is the total number of grammar files in the repository, T_j represents the set of unique types used in the *j*-th textual instance, *m* is the total number of textual instances, and $|\bigcup_{j=1}^{m} T_j|$ represents the total count of unique types across all instances (removing duplicates). This metric indicates how comprehensively the provided instances exercise the defined grammar rules in each repository.

4.6 Step 6: Scenario Judgement

To answer RQ3, which addresses different development scenarios, we determined the used language development scenarios in each repository in Step 6. Xtext supports two language development scenarios, described in Section 2.2. In the grammar-driven scenario, the text definition of the Xtext grammar has a statement starting with the keyword "generate", which results in generating a metamodel from the grammar. When generating the metamodel, Xtext automatically places the metamodel in a folder named "generated". In the metamodel-driven scenario, language developers create a metamodel in a folder they create. We assumed that the developers do not name their created folders "generated", which would be counterintuitive. Considering that there may be multiple Ecore files in a repository, we distinguished three cases in which Ecore files existed in a repository: 1) All Ecore files in the repository are in a folder named "generated", 2) all the folders containing Ecore files in the repository are not named "generated", and 3) some of the folders containing Ecore files in the repository are named "generated" and some are not. We classify the first situation as a grammar-driven scenario, the second situation as a metamodel-driven scenario, and the third situation refers to both scenarios.

4.7 Step 7: Analyze Evolution

To address RQ4, on language evolution, we analyze the evolution and co-evolution in the 226 repositories classified as "Language" by examining the commits of all Xtext/Ecore files and instance files in these repositories. First, we obtained the commits of these files in the 226 repositories using a Python script, including the commit time and message. The commit information was stored in an xlsx table (both the script and the table can be found in our supplemental materials Zhang et al. 2025). Not every commit includes an evolution step in and of itself. For example, changes to a language (as with all software systems) will often be complex enough to warrant effort over multiple days or even weeks, resulting in multiple commits. To study evolution in this paper we therefore introduce a heuristic allowing us to approximate the occurrence of new evolution steps. For that, we calculated the time difference between each commit and the previous commit of the same file. If a commit of a file happens more than 30 days after the previous commit of the same file, then we consider this commit to be the first commit of a new evolution step for that file. This choice is grounded in typical iteration lengths in modern software development processes, particularly Scrum, one of the most widely used methodologies. Scrum iterations (sprints) commonly span 1 to 4 weeks (Paulk 2013; Budacu and Pocatilu 2018; Diebold et al. 2015), with 2 weeks repeatedly mentioned as the most typical duration (Paulk 2013; Budacu and Pocatilu 2018). We argue that if two changes are separated by at least one full iteration, it is natural to treat them as part of distinct development efforts - that is, to consider the later one as part of software evolution rather than the same development surge.

For changes less than 30 days apart, we considered two subgroups of changes: a) Multiple changes committed to the same xtext file within a short time: This would hint at rapid iterations when developing the grammar of the DSL, i.e., the second commit could be the result of lessons learned from the changes made in the first commit. b) Changes in different files committed within a few days: This could indicate that the change is a case of co-evolution, i.e., one of the changes is a reaction to the other change. In both scenarios, we opted for 5 days as a threshold and heuristic to judge whether commits are reactions to each other. Conversely to the justification for the 30-day threshold, the 5-day threshold was chosen to reflect a timeframe within which two changes are highly likely to belong to the same development iteration.

Commits that are the start of a new evolution step were further analyzed with regard to the purpose of the commit. We divided these commits into adaptive, perfective, corrective, and preventive types based on the intentions of software maintenance and evolution described in Section 2.3. To do so we manually analyzed the commit comments of these commits as well as commit comments from iterations, i.e., commits following shortly after the analyzed commit. We marked commits that do not have sufficient information for us to determine their type as "unclear".

As an example, Fig. 3 illustrates the timeline of the creation and changes of three files from the repository JSFLibraryGenerator[6]: an ecore file, a xtext file and an instance file conforming to the grammar. All three files were initially committed in the June of 2015 directly followed by several iterative commits changing the files. In August 2015 an evolution step happened that included a corrective change to the documentation and caused changes in the ecore and instance files. Further evolution changes happened in 2016, January 2017, and

[6] https://github.com/stephanrauh/JSFLibraryGenerator

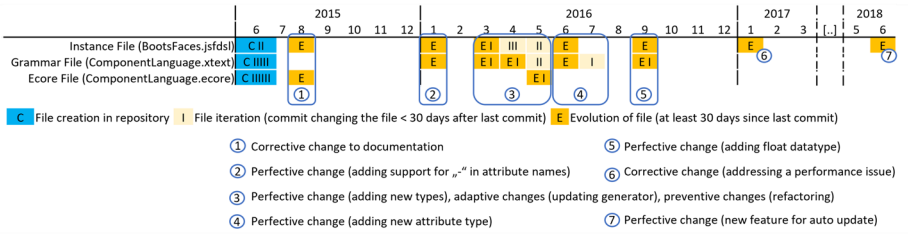

Fig. 3 Example of Evolution History

June 2018. In the last two cases only the instance file changed. The evolution step in March 2016 starts with a simultaneous change of the grammar and instance file. The instance file is then changed over a series of several commits (iterations). The xtext file is iterated on as well, then a break of more than 30 days occurs and the xtext file is changed again, which is counted as a new evolution step in our heuristic. Interestingly, the iterations of the instance file continued during this 30 day period. This example shows that it can be difficult to be sure that a change after more than 30 days really presents an independent evolution step. There are multiple types of changes occurring during that period as well, including perfective, adaptive, and preventive changes. Nonetheless, in most of the cases the heuristic of 30 days allows us to approximate developmental phases, which we consider here to be evolution.

As can be seen in Fig. 3 some evolution steps of a pair of grammar and instance files (and ecore file) occur within days or even within the same commit. To better understand the prevalence of co-evolution of grammars and instances, we conducted a separate investigation on commits starting an evolution step (i.e., commits that changed a grammar or instance file that has last been changed at least 30 days earlier). For each commit, we analyzed whether a file of the other type (instance file or xtext file) in the same repository has been changed within the same commit or within five days (as described above). If so we checked whether the change of that other file presents the start of an evolution step as well (i.e., is not an iteration following other changes). In our example, there is a co-evolution of grammar and instance file in March of 2016. However, the second evolution step of the xtext file in April of 2016 is here not considered a co-evolution with the instance file, even though it changed in close time proximity, since the instance file underwent a continuous iteration starting March of 2016.

5 Results

This section presents the results of our investigation across different repository subsets. In this research, an ample amount of data is collected and analyzed, in both manual and automated ways, with our analysis focusing on different subsets depending on the research question. The automated way used scripts developed by the authors. The resulting dataset (spreadsheet) and our analysis scripts are available from the associated artifact (Zhang et al. 2025). In this section, we will introduce the counts of repositories we obtained by filtering under different conditions in different steps. There are multiple such counts, and we depict the relationship between different counts in Fig. 4 for easy understanding.

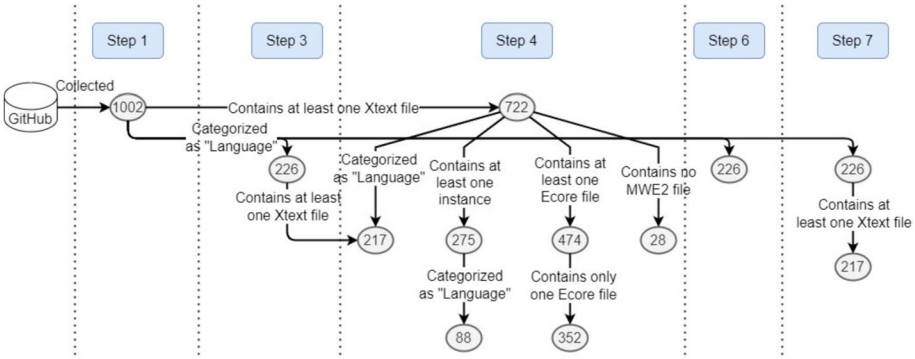

Fig. 4 Count of repositories in different subsets across analysis steps

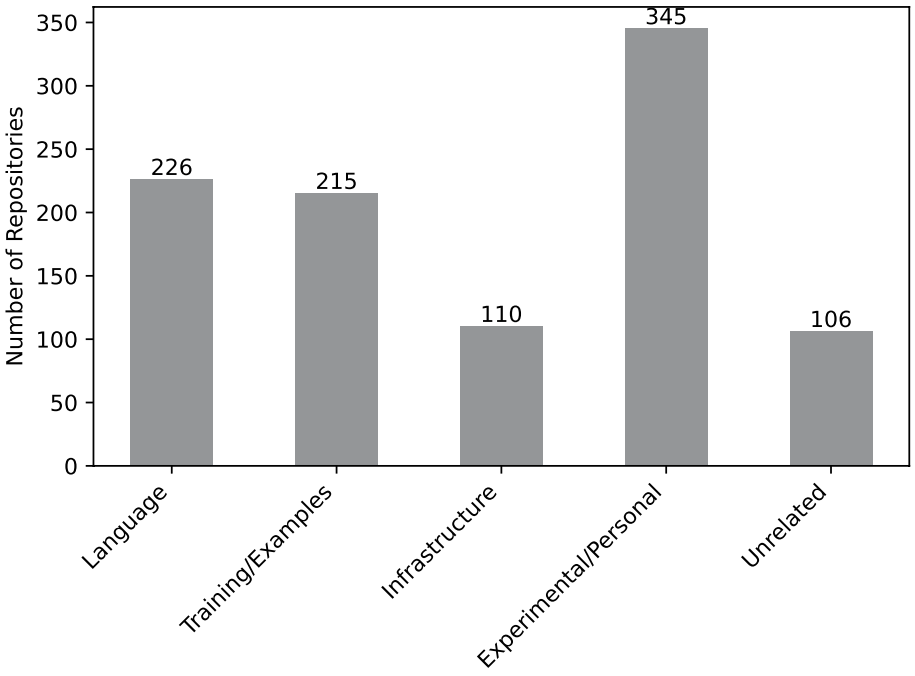

Fig. 5 Classification of repositories

5.1 RQ1: Are There GitHub Projects That Use Xtext? Which are These Projects?

5.1.1 RQ1.1: How Can These Repositories be Categorized?

To overview the 1002 repositories identified via our search methodology, we show the outcome of our manual classification, according to the methodology explained in Section 4.3. As indicated in Fig. 5, we found 226 repositories in which languages have been developed and contain descriptions of them. There are 215 repositories with documented training and

example materials. 110 repositories provide infrastructure to support development with Xtext. Repositories classified as *experimental/personal* are the most numerous, with 343 cases. Additionally, 106 repositories have no relationship to Xtext except for naming.

We illustrate the different categories with examples. In our corpus, *Languages* developed with Xtext span a large variety of cases, from DSLs for specific application domains such as games (`Casino`), telemedicine (`telemed`), document management (`Xarchive`), to technical domains such as JSON schema (`xtext-json`), Quantum computing (`Quingo/compiler_xtext`) and Eclipse launch configurations (`lcdsl`). A noteworthy sub-category, are cases of retrofitting existing languages, such as Oberon (`Oberon-XText`) or GraphQL (`graphQL-xtext-grammar`). *Training/Example* projects comprise tutorials such as 15-minute Xtext tutorial in Chinese (`xtext_tutorial_15_min_zh`). *Infrastructure* projects include technology for integrating Xtext and particular languages in specific contexts, e.g., editors (`vim-xtext`) and build processes (`gradle-xtext-generator`). *Experimental/personal* code often involves a dump of the user's personal workspace (e.g. `Xtext Workspace`). *Unrelated* repositories generally result from a name clash, such as using the name 'xtext' for some unrelated tool (e.g., `resolved`'s text displaying tool `xtext`), or within some longer name, such as `TopXTextUI`.

As useful meta-information for users of our dataset, we collected the number of forks and stars for all repositories and included them in our dataset (Zhang et al. 2025). On GitHub, stars are intended to indicate user interest or endorsement of a repository (GitHub Docs 2025b), while forks represent an explicit act of reuse or experimentation with its contents (GitHub Docs 2025a). As such, both can act as a partial indicator for practical usage. We found that 218 repositories had forks and 321 repositories had stars. The top repositories in either category are main projects for xtext (746 stars; 314 forks) and the associated JVM language xtend (101 stars; 53 forks), as well as popular training examples. As a notable mention, `applause` is a DSL for the cross-platform development of mobile application (98 stars, 24 forks). Note that stars and forks should not be interpreted as a comprehensive usage measures, but as indicative signals of community engagement. In particular, absence of stars should not be taken as a negative indicator, while presence of stars or forks is a positive one.

5.1.2 RQ1.2: Which Trends Define The Application Domains of These DSLs in Recent Years?

As mentioned in Section 4.3.2, we mapped the domains of the 226 repositories classified under "Language" to the 18 categories, and counted the number of mapped repositories under each category. The obtained results are shown in Table 1. The distribution of the categories is shown proportionally in Fig. 6.

Figure 6 shows that half of the Xtext-based DSLs are used in the domains of "Modeling, Simulation, and Design" (e.g. language `Yaktor` in the repository `yaktor-dsl-xtext` which is for *Data and behaviour modeling*), "Programming Languages" (e.g. language `Quingo` in the repository `compiler_xtext` which is a language for *Quantum programming*), "Data Management and Databases" (e.g. language `xtext-orm` in the repository `xtext-orm` which is for defining general ORM models), and "IoT, Embedded Systems, and Hardware" (e.g. language `JKind` which is for *Embedded Systems Modeling and Verification*). To further demonstrate the vitality of these DSL projects, we analyzed commit

Table 1 Count of mapped repositories under different categories

No.	Category	Count
1	Programming Languages	36
2	Software Development and Engineering	17
3	Games	12
4	Web and Mobile Development	10
5	Modeling, Simulation, and Design	37
6	Data Management and Databases	27
7	Security and Networking	6
8	Artificial Intelligence and Machine Learning	4
9	Business and Enterprise Applications	4
10	Healthcare and Life Sciences	3
11	IoT, Embedded Systems, and Hardware	19
12	Mathematics, Logic, and Scientific Computing	11
13	Testing and Verification	6
14	Content, Information and Document Management	12
15	Cloud, APIs, and Web Services	8
16	Graphical User Interfaces (GUI)	6
17	Questionnaire	2
18	Miscellaneous	6

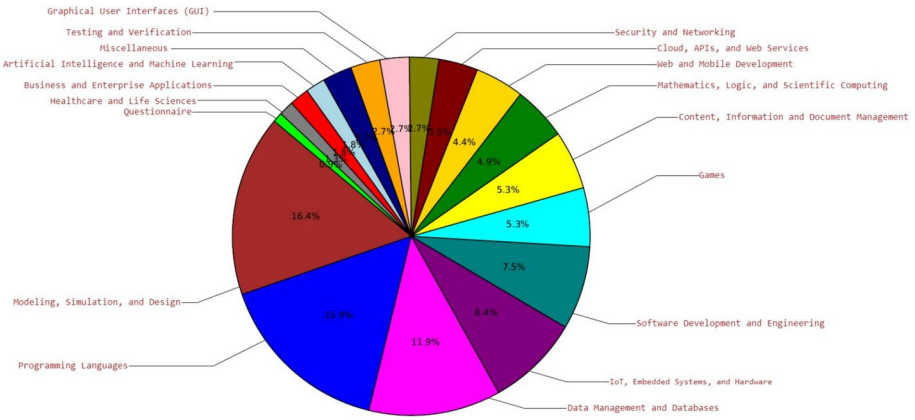

Fig. 6 Proportion of repositories in different categories

activity across the 217 repositories classified as "Language" that contain at least one Xtext file. This analysis reveals 4,793 total commits to language-related artifacts (Xtext, Ecore , and instance files) over the entire duration covered in the dataset, September 2008 through January 2025. Out of these 4,793 commits, 1,722 occurred after January 1, 2020, demonstrating continued development activity in recent years. Additionally, despite its currently increasing relevance, the domain of AI/ML is so far rarely addressed with DSLs built with Xtext.

We obtained the creation time of these 226 repositories, and the results showed that the earliest repository was created in 2010, and the last repository was created in 2023. We observed the change trends of these repositories over a span of 14 years from 2010 to 2023, which are shown in Fig. 7. We found that starting in 2014, Xtext-based language develop-

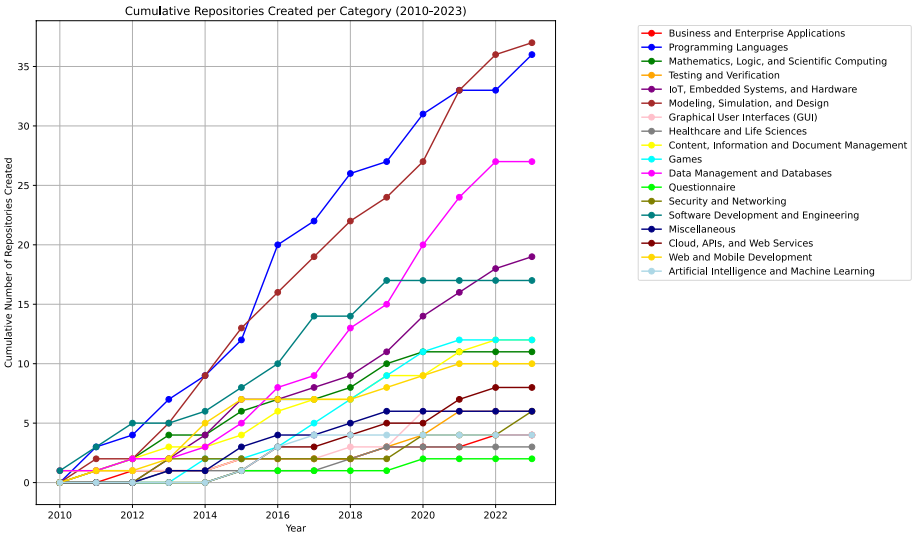

Fig. 7 Cumulative repository created per category (varies with year)

ment entered a period of rapid development, and a large number of repositories were created for developing Xtext-based languages. However, after about 2020, Xtext-based language development entered a period of decline, and the number of repositories under most categories stopped increasing. However, the three categories with the largest proportion, i.e., "Modeling, Simulation, and Design", "Programming Languages", and "Data Management and Databases", still maintained an increase in the number of repositories they contained. Interestingly, after about 2019, the development of Xtext-based languages related to "IoT, Embedded Systems, and Hardware" became more frequent, while at about the same time, the development of Xtext-based languages related to "Software Development and Engineering" suddenly stagnated.

We also investigated the last commit time of the repositories under these categories. We found that the repositories in nearly half of the categories stopped updating before 2022. The earliest to stop updating were the repositories under the "AI and ML" category, whose last commit occurred in 2017. The categories shown in Fig. 7, where new repositories were still created in recent years, include repositories that are still updated recently. That is, categories such as "Modeling, Simulation, and Design", "IoT, Embedded Systems, and Hardware", "Software Development and Engineering", and "Data Management and Databases" contain repositories that still have commits in 2024.

Results of RQ1: We find five categories of repositories—language, training/examples, infrastructure, experimental/personal and unrelated, and there are 226 repositories classified as "Language". From the perspective of domain, these 226 repositories can be categorized into 18 domain classifications. Among them, nearly half of these 226 repositories are categorized into the "Programming languages", "Modeling, simulation, and design", and "Data management and databases" domains, and they are also the three domains with the fastest development and longest active life span of Xtext-based DSLs.

5.2 RQ2: What Language Artifacts for Xtext-Based DSLs do These Repositories Contain?

5.2.1 RQ2.1: What Main Language Definition Artifacts do These Repositories Contain?

With Xtext grammars being one of the core artifacts of Xtext projects, we checked how many are contained in each repository. The results, shown in Table 2, show that of these 1002 repositories, only 722 really contain at least one xtext file. Nearly all repositories classified as *language* contained an Xtext grammar—217 out of 226 cases. The nine exceptions involved repositories that contained Xtext-based editors for a particular language without providing the underlying grammar, such as `Palladio-Editors-VSCode`, and ports of originally Xtext-based DSLs to other workbenches, such as `eJSL-MPS` (Priefer et al. 2021).

For the 722 repositories that contain at least one Xtext file, Table 2 reports the numbers of other included MDE artifacts. Of these repositories, 248 contain no Ecore metamodel file, while 352 repositories contain one Ecore metamodel file. A total of 6 repositories contain at least 100 Ecore files each. All of these are infrastructure projects that use a larger number of examples for testing and/or demonstration purposes; three of them associated with the official Xtext project. We also counted the number of MWE2 files in the repositories, as shown in the sixth and seventh columns of Table 2. We found that among the 722 repositories that contained at least one Xtext file, all but 28 contained at least one MWE2 file. The vast majority of repositories contained only one MWE2 file, and only one repository contained more than 100 MWE2 files. We also found 222 repositories that contained MWE2 files but no Ecore files. Such a big number of projects that contain an MWE file but no Ecore file can be explained by the fact that in a grammar-driven scenario, Ecore files can be fully automatically generated from the underlying grammar, and it is a common practice to not commit automatically generated artifacts to repositories.

5.2.2 RQ2.2: Do These Repositories Contain Both Grammars and Instances That Adhere to it?

Table 3 shows the results of identifying textual instances in the 722 repositories that contain at least one Xtext file, 447 contain no textual instances at all, 103 contain only one textual instance, and 173 repositories contain at least two textual instances. Interestingly, we found two repositories containing more than 1000 textual instances, namely, `xtext/xtext-monorepo` and `eclipse/xtext`, owned by the Xtext team and the Eclipse Foundation, respectively.

Table 2 File type frequency in projects containing Xtext files

	Xtext		Ecore		MWE	
#Files	#Repo	Perc.	#Repo	Perc.	#Repo	Perc.
>= 100	6	0.83%	6	0.83%	1	0.14%
10 - 99	19	2.63%	8	1.11%	19	2.63%
2 - 9	261	36.15%	108	14.96%	264	36.57%
1	436	60.39%	352	48.75%	410	56.79%
0	/	/	248	34.35%	28	3.88%

Table 3 Instances in repositories that contain Xtext grammars

Count of Instances	#Repos	Percentage
>= 1000	2	0.28%
100 - 999	10	1.39%
10 - 99	30	4.29%
2 - 9	130	18.00%
1	103	14.27%
0	447	61.77%

As shown in Table 3, there are 275 repositories that contain at least one Xtext file and at least one instance file. In Section 4.4, we mentioned that in order to check whether the found instances comply with the grammar in the same repository, we performed a sampling analysis, that is, we randomly selected ten repositories that contain both grammars and instances and checked the compliance. The results of this sampling analysis show that the text instances in these ten repositories all comply with the grammar in the same repository. Thus, we work with the assumption that the found instances conform to the grammars.

The absence of textual instances in a repository does not necessarily indicate that the DSL is unused in practice. To validate this, we examined community engagement metrics for the 138 repositories that are categorized as "Language" and contain Xtext grammars but lack textual instances. We found that 38 (27.53%) of these repositories have been forked by other developers, with 3 repositories having more than 10 forks (maximum: 26 forks). Additionally, 54 (39.13%) have received stars from the community, with 6 repositories having more than 10 stars (maximum: 98 stars). This community engagement is consistent with some DSLs without co-located instances being used in practice, with instances potentially maintained in separate downstream repositories or application-specific projects. This pattern reflects the separation between DSL development repositories and DSL usage contexts commonly observed in software development ecosystems.

5.2.3 RQ2.3: To What Extent are Xtext Grammars Covered by Example Instances Contained in These Repositories?

Among the repositories categorized as "languages", there are 88 repositories that contain both at least one xtext file and at least one instance file. In Step 5, we recreated the Xtext projects contained in some repositories using Xtext 2.36. This resulted in 71 of those 88 repositories where the Xtext projects could be fully resolved and parse instances from the same repository. In the other 17 repositories, we could not recreate the Xtext projects they contained due to technical barriers or missing files, such as other Ecore files that they depended on were not provided. In the 71 repositories where we could parse the instances, we used the Xtext projects to parse instances from the same repository to obtain the count of used grammar rules and calculated the percentage, resulting in the results shown in Fig. 8.

We found that among these 71 repositories, most of them provided textual instances that used more than 60% of the grammar rules. Four repositories provided textual instances, but the textual instances were empty or had nothing to do with the grammar at all. For example, in the repository "Bicycle-Shop", the extension for instance files is defined as "nbs". The repository does provide a "nbs" file, however, the content of the instance does not comply with the grammar.

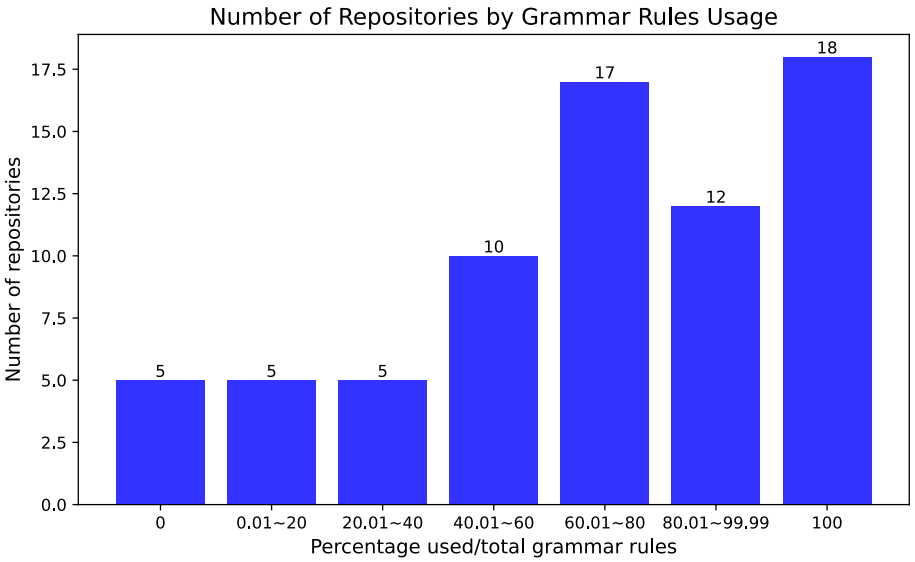

Fig. 8 Number of Repositories by Grammar Rules Usage

Table 4 Frequency of language development scenarios

Scenario	#Repos	#Retrofitting
grammar-driven	167	66
metamodel-driven	41	6
both	9	0
not applicable	9	2

Results of RQ2: We found that 722 repositories contain at least one Xtext file. With the exception of a few, almost all of them contain at least one MWE2 file, and the majority of them contain at least one Ecore file. About a third of them contain at least one instance file. We found that most of the repositories that provide instance files use more than 60% of the grammar rules in the instances.

5.3 RQ3: Development Scenarios

To answer RQ3, regarding development scenarios for language development, we specifically focused on the 226 repositories classified as *language* in RQ1, as these represent fully developed DSL projects, and evaluated their development scenarios as described in Section 4, leading to the results shown in Table 4.

There are a total of 226 repositories classified as *language* , the majority (169) follow a grammar-driven scenario, and 66 of them have developed an Xtext version of an existing language. 41 of the 226 repositories are in a metamodel-driven scenario, and 6 of them have

developed an Xtext version of an existing language. Nine of these 226 repositories contain both grammar-driven and metamodel-driven scenarios, and none of them have developed an Xtext grammar for any existing language. Additionally, the nine repositories that were classified *language* but did not contain Xtext files (described in the results for RQ1), were not suitable for our analysis and hence excluded from it.

We give selected examples for the identified metamodel-driven cases since it is arguably the more complex scenario, involving manual overhead for keeping metamodels and grammars synchronized. `megal-xtext` provides a textual syntax for the MegaL mega-modeling language, a language that by design provides several concrete syntaxes (Favre et al. 2012), a typical motivation for the metamodel-driven scenario. `Kotlin-metamodel` is a repository with the main claim of providing a metamodel for the Kotlin JVM language; the provided Xtext grammar is mentioned as an additional artifact. Other repositories such as `QuestionnaireDSL` do not include a definite explanation for following the metamodel-driven scenario, but at least contain a visualization for the metamodel (*aird* file), which indicates an intention to explicitly design the metamodel. Repositories classified as *both* generally comprise several languages with different workflows, e.g., `telemed` with separate languages for information storage and querying of telemedicine information.

A noteworthy activity that our manual classification of repositories brought forward is *retrofitting*: the implementation of some existing language in Xtext. We found 74 cases of repositories that can be classified as such. In several cases, repositories included implementations of popular practical languages, e.g., GraphQL for graph database querying (`graphQL-xtext-grammar`), JSON Schema for schema definition (`JSON-Schema-to-internal-DSL`), and PlantUML, a notation for lightweight UML diagram creation (`plantuml-eclipse-xtext`). The added value of an Xtext implementation for these languages is to benefit from the editing support offered by Xtext, e.g., automated code completion, syntax highlighting, and formatting. A few cases were concerned with historical programming languages and could be seen as an enthusiastic effort (e.g., `Oberon-IDE`).

The majority of retrofitting cases was developed in a grammar-driven way, which is consequential: The concrete syntax in these cases is fixed and hence, it is natural to specify a grammar matching the existing syntax, thus retrofitting it.

Studying the motivations for retrofitting further is an intriguing topic. For example, retrofitting could be performed as part of a technology stack migration, which would render it a language evolution activity. For the vast majority of repositories, the available descriptions did not include an explicit motivation. In cases involving widely used languages such as UML, Oberon, JavaFX, or bash, describing the changes as language evolution does not seem appropriate. In contrast, for more niche languages—such as `ScytherTool/dsl` for the Scyther security protocol analysis tool—this interpretation might be more plausible. One of the few cases where a motivation was explicitly stated is `RPG-lexer`, where the description reads: *"So this project can be used as an academic project for better understanding of the RPG language but also for better tooling like plug in for 100% free-RPG editors running in i.e. Eclipse with code completion etc."* Overall, we consider retrofitting and evolution to be orthogonal concerns.

We also note that no single language emerged as particularly dominant in terms of retrofitting activity. The most frequently retrofitted languages in our dataset were GraphQL and VHDL, each appearing in three cases.

Results of RQ3: Among the 226 repositories with fully developed languages, while the majority of Xtext-based language development projects involve the grammar-driven scenario, a nonnegligible number relies on the metamodel-driven one. We identify *retro tting*—creating an Xtext implementation of an existing language—as a noteworthy development activity.

5.4 RQ4: How do Xtext-Based Languages on GitHub Evolve Over Time?

5.4.1 RQ4.1: How Can Grammar Evolution in These Projects be Characterized Quantitatively?

To investigate how languages and their artifacts evolve, we focused again on the 226 repositories classified as 'language'.

We were interested in the longevity of language projects and the activity around their included grammars. To study this aspect, we focused on those *language* -classified repositories that contained a grammar, leading to 217 repositories. We observed a time span for updates of the repository and the contained grammar. The time span is calculated as the difference in days between the first and the last commit. We obtained the results shown in Table 5. The results show that 36 repositories were never updated after they were created, while 39 repositories had updates spanning more than 1,000 days, meaning that they were still maintained after a substantial amount of time. Considering the time span of updates to the grammar included in the projects, there are many (i.e. 84) repositories where Xtext files are never updated after they are initially created. There is also a nonnegligible number (i.e. 18) of repositories that still have records of updating Xtext files after a long time (no less than 1000 days).

Furthermore, we considered project lifespan and prevalence of evolution steps. Among the 217 repositories classified as "Language" that contain at least one Xtext file, 76 repositories (35.0%) exhibit evolution steps. Across these 76 repositories, we observe a total of 297 evolution steps, i.e., an average of 3.96 steps per repository, with substantial variation and a maximum of 34 steps in the `epsilon` repository. Focusing on repositories with evolution steps, the average project lifespan is 449.21 days, again with notable dispersion (cf. Table 5 for the broader timespan context).

Next we report on the temporal distribution (volatility) across the lifespan. Evolution activity is not concentrated solely at the beginning of projects: 28% of evolution steps occur in the early phase (≤25% of the project lifespan), 40% in the middle phase (25–75%), and 31.7% in the late phase (>75%). This pattern indicates evolution throughout the lifecycle rather than predominantly front-loaded volatility.

Table 5 Timespans of commits in repositories and Xtext files

	Repo		Xtext	
Timespan (Days)	#Repo	Perc.	#Repo	Perc.
>= 1000	39	17.97%	18	8.29%
100 - 999	54	24.88%	43	19.82%
10 - 99	58	26.73%	44	20.28%
1 - 9	30	13.82%	28	12.90%
0	36	16.59%	84	38.71%

Table 6 Average change to number of lines when comparing first and last commited version of Xtext grammar

Avg. Change to #Lines	#Repos	Percentage
< 0	4	5.02%
0	78	35.62%
> 0 and <= 10	27	12.33%
> 10 and <= 100	68	31.05%
> 100 and <= 1000	33	15.07%
> 1000	2	0.91%

Table 7 Average change to number of rules when comparing first and last commited version of Xtext grammar

Avg. Change to #Rules	#Repos	Percentage
> 100	3	1.37%
<= 100 and > 10	46	21%
<= 10 and > 0	65	29.68%
= 0	91	41.55%
< 0	14	6.39%

We were further interested in how much Xtext grammars change over time. To this end, we considered both the changes to the overall number of lines and the number of rules. Starting with the number of lines, for those 217 repositories that contain at least one Xtext file and are classified as *language* , we compared the first and the last committed version of the Xtext files in terms of their line counts, reporting the averages per repository in Table 6. We can see that the number of lines of text for Xtext grammar has not changed in 35.62% of the repositories. For those repositories that have changed, the vast majority of them contain Xtext files that have basically added the number of lines of text. Among them, the average increased line number of Xtext files in 35 repositories by more than 100.

We also analysed the changes in the number of grammar rules in the Xtext file, leading to the results shown in Table 7. We can see that in more than 40% of the repositories, the number of grammar rules in the Xtext grammar has not changed. Part of the reason is that 66 repositories have not updated their Xtext files since they were initially committed. For those repositories where the number of rules of the Xtext grammar changed, in most of them the number of rules increased over time. In particular, the average number of added grammar rules in the Xtext grammar contained in 47 repositories exceeds ten.

Our dataset (Zhang et al. 2025) contains additional metadata quantifying evolution activities, specifically, the change frequency of grammars, metamodels, and instances per project. While a detailed analysis is outside the scope of this paper, we observe the following trends: As one would expect, grammars are updated more often than metamodels. However, the difference in update frequency is more pronounced in the grammar-driven workflow than in the metamodel-driven one, which might suggest that users in this workflow spend more time polishing concrete syntax aspects. Instances are more likely to be updated in repositories with more Xtext grammar updates.

5.4.2 RQ4.2: How Common are Different Types of Changes During The Evolution of Grammars?

As mentioned in Section 5.2.1, 217 repositories classified as “Language” contain at least one xtext file. Using the same commit dataset described above (4,793 commits from these

217 repositories), we counted the number of days between each commit and the previous commit of the same file. We found that in 192 repositories, the commits to the instance files were never more than 5 days away from the previous commit to the same file. In 115 repositories, the commits to the xtext files were never more than 5 days away from the previous commit to the same file. There is an overlap of 111 repositories in which both, grammars and instances, only show incremental changes with 5 or fewer days between commits. In addition, we found that there were 3809 commits of files that had previously been committed within five days. This hints that DSLs, just like other software, are not carefully designed on paper and then implemented once, but are built in iterations.

We also found that there are 438 commits of xtext, Ecore, or instance files that are more than 30 days away from the previous commit of the same file.[7] These commits come from 78 repositories.

We classified these 438 commits into the four change types described in Section 5.4.2, where we also illustrate each type with examples from our considered projects. We found that about two-thirds of these commits (304 in total) are “perfective commits”. The number of “adaptive” and “corrective" commits is also not small, with 68 and 50 respectively. In addition, there are five commits, only, that belong to the “preventive” type. The proportion of these different types of commits to the total number is shown in Fig. 9. The high number of perfective changes indicates that there is an inherent (functional) need for these languages to evolve. Thus, the changes seen are not just for maintenance, which would be rather associated with corrective and adaptive changes. This indicates that changing requirements on the language are the main driver of DSL evolution.

5.4.3 RQ4.3: How do Textual Instances co-Evolve with Grammar in Real Projects?

Continuing with the 217 repositories analyzed above, we followed step 7 of our methodology, i.e., for those commits files that were more than 30 days away from the previous commit of the same file, we determined whether there was another commit in the same repository that was no more than 5 days away from it but committed a different type of file. For example, in the repository named “gen-angular-grammar” with login “jhonatan89”, the metamodel file “Generator.ecore” was committed on February 2, 2018, we checked if there was another file that was of a different type than the file “Generator.ecore” and it was committed within five days around February 2, 2018. We filled in “YES/NO” in a new column. We found that there were 188 such commits, and they came from 39 repositories.

We checked these 188 commits and the 39 repositories where they were located, and found that 36 of them had a scenario of co-change, i.e., two language artifacts of different file types were changed at the same time or changed one after another within five days, and more than 30 days later, the two files were changed again at the same time or changed one after another within five days. For example, still in the repository “gen-angular-grammar”, the metamodel file “Generator.ecore” and the xtext file “Generator.xtext” evolved simultaneously on November 30, 2017, and they evolved simultaneously again on February 2, 2018. Among these 36 repositories, 33 of them had metamodels and Xtext files that changed simultaneously or within five days of each other, eleven of them had Xtext files and

[7] Commits including two or more xtext, Ecore, or instance files that have previously been committed more than 30 days ago are counted two or more times accordingly.

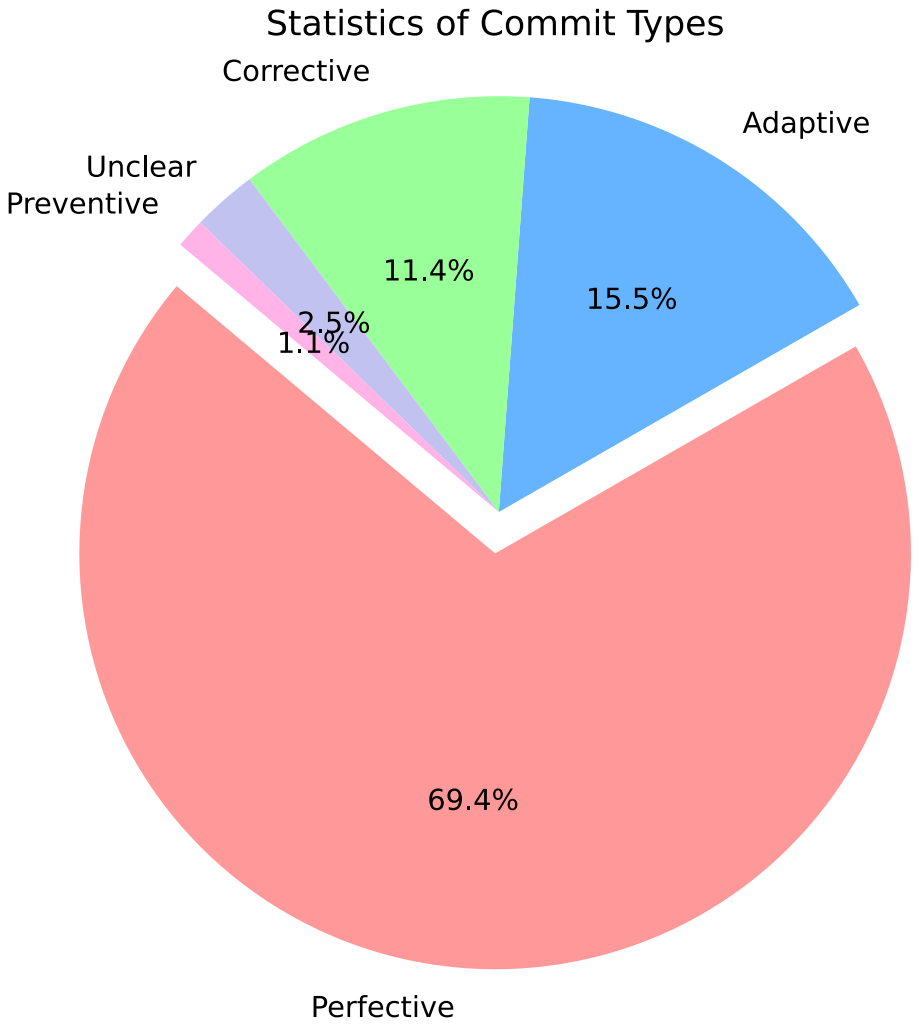

Fig. 9 Statistics of the number of different commit types, i.e., adaptive, perfective, corrective, and preventive commit

instances that changed simultaneously or within five days of each other, and eight of them involved both cases.

By combining these observations with the analysis of commit types, we found that among these 188 commits, there are 151 "perfective" commits, 22 "adaptive" commits, ten "corrective" commits, two "preventive" commits, and another three commits are "unclear". Therefore, compared with the proportion of each commit type in the previous section, the proportion of "perfective" commits here has increased, while the proportion of almost all other commits has decreased.

Results of RQ4: We found a spectrum of project and language development lifespans of Xtext projects on GitHub, with a non-negligible part (8%) of grammars still being updated 1000 days after the initial commit. The amount of change performed in individual languages can be signi cant, with more than 10 rules being added in 22% of all languages. We also extracted 4,793 commits from 217 repositories and found that in about half of the repositories, no commit of Xtext les and instance les was more than ve days away from the previous commit. Moreover, most cases of evolution were classi ed as "perfective" changes, indicating changing requirements on the DSLs. We also found co-evolution of les of different le types (instance, grammars, and metamodels) in 36 repositories. In addition, we also found that the evolution steps were relatively evenly distributed in different phases of the project, rather than concentrated in a speci c phase.

6 Discussion

We will discuss our results from three perspectives: the need for approaches to support co-evolution in textual DSLs, the need for a better understanding of DSL development in practice, and threats to the validity of our research.

6.1 Need for Approaches for co-Evolution in Textual DSL Evolution Contexts

Our analysis in RQ3 reveals the existence of DSL evolution projects that are maintained over several years and involve significant changes to the language over time. In such projects, challenges arise from the synchronized evolution of all involved artifacts, including grammars, metamodels, instances, and workflow files. In a general MDE context, synchronized evolution is a well-studied area, termed *co-evolution* or *coupled evolution*. A plethora of existing work (García et al. 2012; Khelladi et al. 2017; Kusel et al. 2015; Vaupel et al. 2015; Kessentini and Alizadeh 2020; Di Ruscio et al. 2023, 2013) provides foundational approaches for managing co-evolution between metamodels and other artifacts, typically models and transformations. However, there remains a significant gap in tool support and methodologies specifically tailored to the co-evolution of artifacts involved in textual DSLs developed with frameworks like Xtext, with their focus on both metamodels and grammars.

Our observations hint on a number of requirements that need to be taken into account when approaching the co-evolution problem for textual DSLs:

- Change and specifically evolution of DSLs is not just about maintenance activities, as witnessed by the majority share of "perfective" changes in cases where grammars and instances change after more than 30 days (see Fig. 9). In our corpus, treating DSLs as evolving software systems appears warranted; tool support should therefore consider functionality-changing evolution alongside maintenance.
- Rapid iterations of the DSL happen. We observe for 106 of all repositories that are classified as languages changes in instances, metamodels, and grammars within less than 5 days. Frameworks supporting textual DSLs (e.g., Xtext) would benefit from first-class support for such rapid iterations.

- Both grammar-driven and metamodel driven development happens and is subject to evolution. Frameworks need to support both evolution cases.
- Co-evolution between instances and grammar happens and our numbers likely underestimate the frequency. In 39 out of 275 repositories with grammars and instances we observed that both grammars and instances change together after a break of more than 30 days, hinting on co-evolution. However, since instances in the DSLs repository are likely just example and serve testing and documentation purposes, this number does not include all cases of co-evolution needed for productive instances of the DSLs, which are not seen in these repositories.

To address some of the complexity in the metamodel-driven scenario (RQ3), one could rely on principles from the grammarware sphere and support an operator-based approach to grammar evolution (Zaytsev 2014). In our recent work, we follow up on this idea to automate parts of the synchronization of the grammar after metamodel changes (Zhang et al. 2023a, 2024, 2023b; Zhang 2023). Another open challenge is grammar-instance co-evolution, which could benefit from the available foundational approaches for metamodel-model co-evolution, but needs to deal with concrete syntax aspects of grammars. This problem has not been addressed yet in the technical space of Xtext. Lämmel's LAL approach (Lämmel 2016) could be useful for validating a solution that addresses it.

Future research should focus on integrated approaches and tools that facilitate the simultaneous evolution of all related DSL artifacts. The comprehensive dataset compiled from our study, particularly the 196 cases where all crucial artifacts such as grammar, metamodel, and instances are available, presents a valuable resource for developing new co-evolution approaches, by supporting their design, testing and evaluation.

6.2 Need for A Better Understanding of DSL Development in Practice

Several observations show that we need a better understanding of how DSL development looks like in practice in order to better support it. This concerns practices such as testing and documentation, versioning support, and the accessibility to domain experts outside computer science.

6.2.1 What Testing and Documentation Practices are Used for The DSLs' Development?

While some of the studied repositories that are classified as languages include instances, it is only a minority. Even among those repositories with instances, the majority does not include a sufficient variety of instances to cover all grammar rules. Thus, in only 18 cases, we found instance examples covering the complete grammar of the DSL. This leads to the question how these DSLs are tested and documented. Instances are important for testing a language. They also play a key role in supporting learnability, which is widely considered an important aspect of usability (International Organization for Standardization 2018). In the context of DSLs, examples are recommended both in expert guidelines (Völter 2009) and in empirical studies, which show that novice users frequently start by modifying existing examples when learning a language (Rennels and Chasins 2023). Similarly, language documentation, e.g., tutorials, rely on example instances, the most famous being variations of "Hello World" that are used to introduce language users to basic language concepts and allow them to

get started learning a language. Thus, it is unclear how testing and documentation through example instances is done for most of the languages in the found repositories. On the other hand, it is possible that the instances are stored outside of the DSL's main repository. This would be explainable with the architectural setup of Xtext, in which language developers use separate Eclipse instances and associated workspaces to edit and to test the language definition. Hence, language definition artifacts (grammars and metamodels) are stored in different workspaces than example cases, and language engineers need to invest dedicated effort in order to make available example cases together with the language definitions. Still, this hints that there is a difference to the practices in other open source software systems, where tests are often stored in the same repository as the code and documentation is at least linked from the repository.

Beyond the documentation and instance coverage issues identified above, our analysis reveals a methodological limitation: we did not systematically examine formal testing practices for DSL development. Modern software engineering emphasizes comprehensive testing strategies, including unit tests, integration tests, and validation rules, yet our repository analysis focused primarily on core DSL artifacts (grammars, metamodels, and instances). This means we cannot determine the extent to which Xtext-based DSL projects employ formal testing methodologies or maintain comprehensive test suites. Given the critical role of testing in maintaining software quality and facilitating evolution—particularly relevant given our findings on frequent DSL changes—investigating DSL testing practices represents an important area for future research. Understanding how DSL developers ensure code quality and reliability could inform the development of better tooling and methodologies specifically tailored for DSL testing and validation.

6.2.2 Is There Missing Support to Version Xtext Projects?

In RQ1, we found that out of the 280 repositories that did not contain Xtext files, a number of them contained other MDE artifacts. Specifically, 9 repositories contain at least one Ecore metamodel but no Xtext files. Possible explanations for these cases are: First, the repository owners were aiming at a metamodel-driven scenario, but did not complete the definition of the grammar. Second, the repository owners created the Xtext files from an Ecore metamodel, and generated them into a different directory that was not included in the repository. Similarly, three of these repositories contain MWE files and two contain instance files, however, these repositories are often incomplete, as can be discovered by importing them into Eclipse, which leads to error messages. This observation goes hand-in-hand with the observation above that test instances require other Eclipse instances and, thus, other workspaces, which are likely to not be versioned together with the DSL itself. Future work needs to investigate whether the architecture of eclipse/Xtext makes it more difficult to access proper version management and establish the requirements for more suitable version management systems to support DSL development.

6.2.3 Is Xtext Only Used for DSLs Built for Software Developers?

The distribution of DSL domains reveals a strong focus on domains where the expected user are computer scientists or software developers themselves. Only surprisingly few of the DSLs are from other domains, e.g., Healthcare and Life Sciences, or DSLs for develop-

ment of questionnaires. This is remarkable, as one purpose of DSLs is to make programming accessible to domain experts that are not classical programmers. The results could be explained in two ways: It is, of course, possible that these results are representative of how DSLs are distributed. In this case, it would be worthwhile investigating why DSLs are not used or worth using in other domains. However, more likely, the results are due to our scope, that is, our focus on open-source DSLs built with Xtext. First, our focus on GitHub might be a factor of bias, as for non-PL domains, there might be a stronger impact of business considerations that prevent an open source publication of DSLs. Second, DSLs for domains outside computer science could be inherently less likely to be built with Xtext. The question is whether the same might hold for other DSL frameworks, like Langium (GmbH 2024), or whether those DSLs are mostly built without the use of language frameworks. This would imply that these DSLs are potentially even more vulnerable to issues like language evolution, a topic addressed in research for DSLs build on concepts such as MOF-like metamodels (Hebig et al. 2016a). These observations raise an important question for the language engineering community – particularly those developing DSL tooling such as Xtext and, to the extent our findings apply to them, other workbenches – regarding the accessibility and applicability of our frameworks to engineers and researchers in domains beyond software development. Future work is needed to answer the question how DSLs for non-computer-science domains are built and investigate the consequences of that.

A related pattern emerges when considering the longevity of DSL projects. Among the repositories that remain active in recent years, a noticeable concentration of programming-language-like DSLs can be observed. This aligns with the technical orientation of the domains discussed above, but also raises further questions. Specifically, we see two plausible explanations for this pattern. First, it may reflect the nature of Xtext as a language workbench that is particularly well-suited for DSLs with rich, structurally complex syntax, which are commonly found in programming-language-like DSLs. Such DSLs benefit more directly from Xtext's advanced editor support and validation infrastructure, increasing the incentive to maintain them over time. Additionally, these DSLs are often central infrastructural components in larger software ecosystems, which further contributes to their sustained development. Second, this pattern may signal a broader limitation in tooling sustainability. DSLs with simpler or less programming-like syntax - typical for application-specific or non-technical domains - may receive less long-term support not due to lack of utility, but due to a mismatch between their needs and the capabilities or complexity of available language workbenches. This observation suggests a need for tool developers to more explicitly consider sustainability and usability concerns across a broader range of DSL types. We reflect this in our conclusion as a key direction for future research.

6.2.4 What is The Compliance Relationship Between Artifacts in Practice?

Our comprehensive dataset paves the way for future research on language development and evolution, especially for detailed studies of language evolution extended on RQ4. In this paper, we have conducted preliminary research on the co-changes of different language artifacts in the same repository, but have not yet analyzed the actual compliance relationship between these language artifacts. For example, we found many cases where Ecore files and Xtext files co-evolve in the same repository, but it is not clear whether the Xtext file complies with the Ecore file and vice versa. Finding repositories with such scenarios of co-

changes of different artifacts allows us to focus future work on a smaller set of repositories. In addition, this repository collection can help us find suitable case languages more quickly for developing and studying tools for grammar and instance co-evolution, ultimately fostering more robust and adaptable languages.

6.2.5 Broader Relevance to Textual DSL Development

The grammar-instance co-evolution patterns we documented in Xtext-based projects appear to represent broader challenges in textual DSL development. Examination of selected repositories using other language development technologies reveals comparable coordination challenges between language definitions and their instances. For example, the repository "DescribeML"[8] develops a DSL based on *Langium*, whose grammar evolves on Feb 16, 2023, and its corresponding instances also evolve at the same time. This demonstrates that the fundamental challenge of maintaining consistency between evolving language specifications and existing instance files transcends specific technological implementations. This suggests that the co-evolution patterns and maintenance challenges we identified in the Xtext context reflect more general phenomena in textual DSL development.

Similarly, the retrofitting phenomenon we observed — where existing languages are implemented in new language workbenches — represents a broader trend in language ecosystem evolution. Evidence of comparable retrofitting activities can be found across different language development technologies, as seen in repositories such as "vscode-sysml-v2"[9], its authors implemented the language SysML in *Langium*, which is a typical retrofitting activity. There is a similar situation in *textX*, such as in the repository "usysml-textx"[10], its authors implemented a subset of the language SysML in *textX*.

These observations, combined with our systematic repository mining methodology, suggest that our analytical approaches and findings about DSL development challenges have relevance beyond the Xtext context, providing directions for future comparative studies across different language workbench ecosystems.

Beyond the transferability of our findings, we also see potential for adapting our methodology to other language frameworks. While our study focused on Xtext-based projects, the underlying approach—mining repositories, classifying artifacts, identifying instance usage, and analyzing co-evolution via commit history—relies on patterns that are not tied to a specific framework. Adapting the method to other technologies, such as *Langium*, *textX*, or classical tooling like *ANTLR*, would primarily require adjusting the file identification logic (e.g., targeting `.g4` files for ANTLR), instance detection strategies, and possibly workflow inference mechanisms. The core structure of our analysis remains applicable, supporting future comparative studies across diverse DSL ecosystems. Some research questions, such as artifact availability (RQ2) and co-evolution dynamics (RQ4), are broadly transferable. Others, such as the development scenarios in RQ3, are more specific to Xtext, which involves managing both a grammar and a metamodel that may co-evolve and require synchronization. In frameworks without this dual-artifact structure, this aspect of RQ3 may need to be reformulated or dropped.

[8] https://github.com/SOM-Research/DescribeML

[9] https://github.com/EDKarlsson/vscode-sysml-v2

[10] https://github.com/igordejanovic/usysml-textx

6.2.6 What is The Impact of Tool Evolution on DSL Development?

Our study focuses on DSL language evolution without considering the evolution of underlying language workbench tools themselves. Tool evolution, such as major version transitions in DSL frameworks, represents an important dimension of complexity that can force developers to migrate or recreate their DSLs (Brambilla and Fraternali 2014). Industrial experiences have shown that maintaining DSL projects across tool versions requires careful planning for extensibility and automated migration support from the beginning of DSL design (Brambilla and Fraternali 2014). Understanding how tool evolution affects DSL development practices and what support mechanisms are needed for smooth DSL migration across tool versions represents an important area for future investigation.

6.3 Threats to Validity

Threats to internal validity arise from us relying on the GitHub API. Since we cannot access the implementation of the GitHub API, we cannot verify *completeness*: the implementation could be inexact, which could lead to repositories not being captured by our query. As a safeguard, we checked whether a total of three expected repositories personally known to us appeared in our results, which was the case. Nevertheless, anecdotally, a colleague to whom we made available our dataset informed us about a project that was not part of it. Still, our findings that arise from a substantial number of cases and highlight the existence of understudied phenomena—metamodel-based evolution, retrofitting, long-living Xtext projects–do not require completeness to be valid.

Furthermore, repositories can be duplicates of each other, which might bias the results. For our *language* -classified repositories, which we investigate in RQ2 and RQ3, we checked that this is not the case and can exclude duplicates. To this end, we relied on the assumption that duplicate repositories will generally share the same name, as they are named after the contained language. For those cases where repositories indeed had the same name, we manually validated that they are indeed not duplicates. For the *experimental/personal* category, which generally does not make any statements about the quality about the included repositories, anecdotally, some repositories have the same contents as others, potentially arising from having followed the same tutorial with the basic Xtext examples.

In addition, in Step 5, we manually judged the domain of the repositories and the categories they belonged to. Potential bias posed an internal threat, so for each data, we used another person's review to reduce the bias of manual judgment. Similarly, we manually judged the type of commits in Step 7, which may also be biased. We had a second person to take a sample review to reduce bias.

Threats to external validity Considering external validity, a limitation of our study is that patterns extracted from GitHub cannot be assumed to generalize to all language engineering contexts. In particular, organizational or proprietary DSLs may follow different evolution dynamics that are not visible in our dataset. Our findings therefore reflect how DSLs are created and evolved in open-source repositories on GitHub, and their relevance for other environments must be interpreted with caution. At the same time, GitHub provides a uniquely systematic and reproducible basis for large-scale analysis, making it an appropriate and valuable source for identifying recurring practices and challenges in an area of DSL evolution.

Furthermore, our scope is restricted to Xtext, a widely used technology in the MDE community. There is a larger variety of existing language workbenches (Erdweg et al. 2015), not all of which might equally benefit from our findings. The results from our study could be transferable to other workbenches that use a similar strategy for separating abstract and concrete syntax specification like Xtext, such as textX and langium. Yet, transferring our results to language workbenches that employ an entirely different paradigm (e.g., in the case of MPS, projectional editing) might be infeasible. We further explore questions of transferrability of our results elsewhere in this section, e.g., in Sections 6.2.3 and 6.2.5.

Additionally, our analysis is restricted to artifacts co-located within the same repository, leaving an assessment of instance availability (RQ2.2) and co-evolution patterns (RQ4.3) in separate downstream repositories outside our scope. Our finding that two-thirds of language repositories lack textual instances may reflect our analysis scope, as instances could be maintained separately in private repositories or enterprise applications not captured by our approach. However, in that case, they would be harder to discover by novice users in the motivating scenarios for our scoping decision.

Threats to construct validity Considering construct validity, to mitigate the impact of subjectivity on our classification, we extensively discussed the criteria and problematic cases and eventually found consensus for all of them. Our classification further relies on documentation provided by the repository owners, which might not always be accurate or complete, leading to two consequences: First, we did not investigate whether repositories were from industry or academia, which generally was infeasible to tell from the documentation. Second, some of our *language*-classified repositories, in particular, among those classified as *retrofitting*, might be exclusively intended for self-teaching or demonstration purposes, but this context is unavailable to us. Users of our dataset are advised to use it in a way that makes sure that their assumptions are met, for example, taking into account our collected change history meta-information to identify cases with a rich evolution history. Another threat that could explain the dominance of software development-related DSLs is our search focus on GitHub. It is possible that we systematically miss out on other types of DSLs, as these might not be hosted there, e.g., due to business considerations.

In addition, a threat to construct validity exists in our approach to identifying DSL evolution through commit time intervals. We use time intervals between commits (more than 30 days for evolution, fewer than five days for co-evolution changes and rapid iterations on Xtext files) as a proxy for identifying evolutionary changes. However, this measurement approach may not fully capture the actual nature of DSL evolution. Depending on the process stage (e.g., during maintenance) evolutionary changes could occur within short time intervals (< 30 days), leading to false negatives. Conversely, long intervals between commits (>30 days) might not necessarily represent true evolutionary changes, potentially resulting in false positives. This threatens the construct validity of our study as the temporal proximity of commits may not always accurately reflect the nature of the changes made to the DSL. To address this threat in our future work, we plan to combine multiple evolution indicators beyond commit intervals, including qualitative analysis of change types, expert reviews of the modifications, and in-depth case studies to establish more robust evolution identification criteria. In particular, distinguishing between semantically minor changes

(e.g., metadata edits) and more substantial ones (e.g., grammar refactorings) would provide valuable nuance. This could be supported by fine-grained, possibly AST-level, change analysis to better capture the scope and impact of individual evolution steps.

Furthermore, our analysis focused solely on commits in the master branch, without examining branching strategies and pull request workflows. This represents a threat to construct validity, as some evolution activities may occur in feature branches, client-specific branches, or collaborative workflows, and evolutions merged via squashed commits may not be visible. Conversely, master-branch histories may also include commit sequences shaped by prior branch-level development, meaning that some activities are reflected in a less granular form. An additional analysis of the 226 repositories considered in the evolution study (RQ4) indicates that branching activity was generally low (mean: 2.11 branches, median: 1.0, with only 4.9% of repositories having more than five branches), though notable exceptions such as `yaktor-dsl-xtext` (37 branches, mostly release branches) and `MetaCrySL` (17 branches, mostly feature branches) demonstrate that more extensive branching is possible. Taken together, these observations imply that our findings characterize DSL evolution as captured on the master branch of GitHub-hosted Xtext projects, which we consider the canonical evolution history. Future work could complement this perspective by analyzing collaborative development patterns in branches and pull requests.

Additionally, our file search methodology identifies artifacts by current location rather than tracking file path changes over time, which could lead to some evolution steps being missed by our analysis. However, we argue the impact of this limitation is likely minimal for two reasons: First, large-scale refactoring involving file relocations appears to be relatively rare in the DSL repositories we examined, as most follow standard Xtext project structures. Second, our analysis focuses on identifying patterns and trends across a large dataset rather than achieving complete coverage of every evolution step. Missing some evolution activities due to file relocations would not fundamentally alter our key findings about development scenarios, artifact co-evolution patterns, or the prevalence of different change types.

Two threats arise in the context of our classification efforts. First, our use of LLMs for a specific step in our characterization of domains for RQ1.2 – namely, the identification of categories to group our manually identified domains – leads to a threat of potential accuracy issues. We describe the mitigation of this threat in Section 4.3.2. Second, during our manual labeling of evolution steps for RQ4.2, we might have mislabeled some of the observed *perfective* changes, as the improvement (which we observed from commit message and code differences) could actually be the consequence of a previous misunderstanding of the domain, which then would make a characterization as *corrective* more appropriate. Such cases would be infeasible for us to detect, as we cannot access that context.

7 Conclusion

This study provides the first large-scale empirical investigation into the development and evolution of Xtext-based DSLs on GitHub, analyzing 1002 repositories and manually classifying 226 of them as containing fully developed languages. We addressed four research questions, covering repository characteristics, artifact availability, development scenarios, and patterns of evolution and co-evolution. We formulate the following key insights:

- In our corpus, Xtext-based DSLs are predominantly developed by and for software engineers, with only limited adoption in non-programming domains. This challenges the long-held goal of DSLs as tools for empowering domain experts beyond computer science.
- In our data, grammar-driven development remains the dominant workflow, but a significant minority of projects still rely on metamodel-driven approaches, underscoring the need for improved co-synchronization support.
- Within our corpus, retrofitting of existing languages into Xtext is a frequent practice, indicating that DSL frameworks are not only used to create new languages but also to re-implement and maintain legacy or external ones.
- Across the analyzed repositories, evolution is frequent and ongoing, with many DSLs undergoing multiple substantial changes over time. Most evolution activities are perfective, rather than corrective or adaptive, suggesting that DSLs are actively extended rather than merely patched.

From our insights, we derive the following recommendations to tool builders and practitioners:

- Tooling should prioritize evolution support, particularly for synchronizing grammar, metamodel, and instance files. Co-evolution remains largely manual, especially for textual DSLs such as Xtext.
- Frameworks should improve accessibility and onboarding, particularly for users outside the programming domain. This may include reducing technical barriers, improving documentation, and providing templates aligned with domain expert needs.
- Given the frequency of retrofitting observed in our data, frameworks could offer more streamlined workflows, templates, or migration paths tailored to the reimplementation of existing languages.

We foresee the following directions for future work. First, understanding DSLs in complimentary contexts, in particular, non-open-source and non-software-engineering ones, is crucial. Our study is limited to public GitHub repositories using Xtext; how DSLs are developed in other domains or using other frameworks (e.g., Langium, ANTLR, textX) remains an open question. Second, automated support for identifying and classifying evolution activities, beyond heuristics based on time or file changes, could enable more accurate tracking of language growth and maintenance needs. Third, investigating retrofitting as a broader phenomenon across language workbenches may shed light on how tool ecosystems can better support incremental adoption and legacy integration. Fourth, analyzing usage patterns in downstream repositories that rely on these DSLs could provide a richer understanding of the practical impact and stability of language artifacts, particularly considering how these repositories evolve in response to changes in the language.

By releasing our dataset and analysis scripts, we aim to support further empirical studies and tool development for Xtext-based and similar textual DSL ecosystems. We hope this work contributes to a more evidence-based foundation for designing, maintaining, and evolving DSLs in practice.

Author Contributions Author One participated in the methodological design, data collection, results analysis, and writing, and developed various scripts required to implement the data collection process. Author Two

proposed the topic of this article and participated in the methodological design, data collection, results analysis, and writing. Author Three participated in data collection and writing.

Funding Open access funding provided by University of Gothenburg. This research received no external funding.

Data Availability The data is fully open in Zhang et al. (2025).

Declarations

Conflicts of Interest The authors declare that they have no conflict of interest.

Ethical Approval This study does not involve any human participants or animals.

Informed Consent Not applicable.

References

Akesson B, Hooman J, Sleuters J, Yankov A (2020) Reducing design time and promoting evolvability using domain-specific languages in an industrial context. In: Model Management and Analytics for Large Scale Systems, Elsevier, pp 245–272

Babur Ö, Constantinou E, Serebrenik A (2024) Language usage analysis for EMF metamodels on GitHub. Empir Softw Eng 29(1):23

Bettini L (2016) Implementing domain-specific languages with Xtext and Xtend, second edition edn. Packt Publishing Ltd., Birmingham, UK, first published: August 2013, Second Edition: August 2016

Brambilla M, Fraternali P (2014) Large-scale model-driven engineering of web user interaction: The webml and webratio experience. Sci Comput Program 89:71–87

Budacu E, Pocatilu P (2018) Real-time agile metrics for measuring team performance. Inf Econ 22(4):71–79

Chen B, Yi F, Varró D (2023) Prompting or fine-tuning? a comparative study of large language models for taxonomy construction. In: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, pp 588–596

Damasceno CDN, Strüber D (2021) Quality guidelines for research artifacts in model-driven engineering. In: MODELS, pp 285–296

Denkers J (2024) Domain-specific languages for digital printing systems. Phd thesis, Delft University of Technology

Deursen AV, Klint P (1998) Little languages: little maintenance? J Softw Maint Res Pract 10(2):75–92

Di Ruscio D, Iovino L, Pierantonio A (2013) A methodological approach for the coupled evolution of metamodels and ATL transformations. In: ICMT, pp 60–75

Di Ruscio D, Di Salle A, Iovino L, Pierantonio A (2023) A modeling assistant to manage technical debt in coupled evolution. IST 156:107146

Diebold P, Ostberg JP, Wagner S, Zendler U (2015) What do practitioners vary in using scrum? In: Proceedings of the 16th International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2015), Springer, Helsinki, Finland, pp 40–51

Erdweg S, Van Der Storm T, Völter M, Tratt L, Bosman R, Cook WR, Gerritsen A, Hulshout A, Kelly S, Loh A et al (2015) Evaluating and comparing language workbenches: Existing results and benchmarks for the future. COMLAN 44:24–47

Favre JM, Lämmel R, Varanovich A (2012) Modeling the linguistic architecture of software products. In: MODELS, pp 151–167
García J, Diaz O, Azanza M (2012) Model transformation co-evolution: A semi-automatic approach. In: SLE, pp 144–163
GitHub Docs (2025a) About forks. https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/about-forks, accessed: 2025-10-02
GitHub Docs (2025b) About stars. https://docs.github.com/en/rest/activity/starring, accessed: 2025-10-02
GmbH T (2024) Langium. https://langium.org
Gousios G, Spinellis D (2017) Mining software engineering data from GitHub. In: ICSE Companion, pp 501–502
Hebig R, Khelladi DE, Bendraou R (2016) Approaches to co-evolution of metamodels and models: A survey. IEEE TSE 43(5):396–414
Hebig R, Quang TH, Chaudron MRV, Robles G, Fernandez MA (2016b) The quest for open source projects that use UML: mining GitHub. In: MODELS, p 173–183
Herrmannsdoerfer M, Benz S, Juergens E (2009) COPE-automating coupled evolution of metamodels and models. In: ECOOP, pp 52–76
International Organization for Standardization (2018) ISO 9241-11:2018 – Ergonomics of human-system interaction – Part 11: Usability – Definitions and concepts. https://www.iso.org/standard/63500.html, accessed: 2025-07-06
Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: MSR, pp 92–101
Kessentini W, Alizadeh V (2020) Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search. In: MODELS, pp 68–78
Khelladi DE, Rodriguez HH, Kretschmer R, Egyed A (2017) An exploratory experiment on metamodel-transformation co-evolution. In: APSEC, pp 576–581
Kögel S, Tichy M (2018) A dataset of EMF models from Eclipse projects. https://doi.org/10.18725/OPARU-9850
Kosar T, Bohra S, Mernik M (2016) Domain-specific languages: A systematic mapping study. IST 71:77–91
Kusel A, Etzlstorfer J, Kapsammer E, Retschitzegger W, Schwinger W, Schönböck J (2015) Consistent co-evolution of models and transformations. In: MODELS, pp 116–125
Lämmel R (2016) Coupled software transformations—revisited. In: Software Language Engineering (SLE), pp 239–252
Lämmel R (2018) Software Languages: Syntax, Semantics, and Metaprogramming, 1st edn. Springer Cham, Cham, https://doi.org/10.1007/978-3-319-90800-7
Lientz BP, Swanson EB (1980) Software maintenance management. Addison-Wesley Longman Publishing Co., Inc
López JAH, Cuadrado JS (2022) An efficient and scalable search engine for models. Softw Syst Model 21(5):1715–1737
López JAH, Cánovas Izquierdo JL, Cuadrado JS (2022) ModelSet: a dataset for machine learning in model-driven engineering. Softw Syst Model pp 1–20
Mengerink JG, van der Sanden B, Cappers BC, Serebrenik A, Schiffelers RR, van den Brand MG (2018) Exploring DSL evolutionary patterns in practice: a study of DSL evolution in a large-scale industrial DSL repository. In: MODELSWARD, pp 446–453
Mengerink JG, Noten J, Serebrenik A (2019) Empowering OCL research: a large-scale corpus of open-source data from github. Empir Softw Eng 24:1574–1609
Paik J, Wallin T (2020) How to write better documentation for your open source project. https://opensource.com/article/20/8/documentation-open-source-projects, accessed: 2025-10-04
Paulk MC (2013) A scrum adoption survey. Softw Qual Profession 15(2):27–34
Priefer D, Rost W, Strüber D, Taentzer G, Kneisel P (2021) Applying MDD in the content management system domain. Softw Syst Model 20(6):1919–1943
Radevski S, Hata H, Matsumoto K (2016) Towards building api usage example metrics. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, pp 46–56. https://doi.org/10.1109/SANER.2016.79
Ratiu D, Nehls H, Joanni A, Rothbauer S (2021) Use mps to unleash the creativity of domain experts: Language engineering is a key enabler for bringing innovation in industry. In: Domain-Specific Languages in Practice: with JetBrains MPS, Springer, pp 25–52
Rennels L, Chasins SE (2023) How domain experts use an embedded dsl. Proceed ACM Program Languages 7(OOPSLA2):1499–1530
Robles G, Chaudron MR, Jolak R, Hebig R (2023) A reflection on the impact of model mining from GitHub. Inf Softw Technol 164:107317

Rust Project Developers (2024) Rustdoc: Documentation tests. https://doc.rust-lang.org/rustdoc/write-documentation/documentation-tests.html, accessed: 2025-10-04
Schuts M, Alonso M, Hooman J (2021) Industrial experiences with the evolution of a dsl. In: Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling, pp 21–30
Shrestha SL, Boll A, Chowdhury SA, Kehrer T, Csallner C (2023) Evosl: a large open-source corpus of changes in simulink models & projects. In: MODELS, pp 273–284
Stahl T, Völter M (2006) Model-driven software development: technology, engineering, management. John Wiley & Sons
Steinberg D, Budinsky F, Merks E, Paternostro M (2008) EMF: Eclipse Modeling Framework. Pearson Education
Thanhofer-Pilisch J, Lang A, Vierhauser M, Rabiser R (2017) A systematic mapping study on DSL evolution. In: SEAA, pp 149–156
Tratt L (2008) Evolving a DSL Implementation, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 425–441. https://doi.org/10.1007/978-3-540-88643-3_11
Vaupel S, Strüber D, Rieger F, Taentzer G (2015) Agile bottom-up development of domain-specific IDEs for model-driven development. In: FlexMDE, pp 12–21
Völter M (2009) Best practices for dsls and model-driven development. J Object Technol 8(6):79–102
Völter M, Benz S, Dietrich C, Engelmann B, Helander M, Kats L, Visser E, Wachsmuth G (2013) DSL Engineering - Designing, implementing and using domain-specific languages. M Volter / DSLBook. org, nEO
Wachsmuth G (2007) Metamodel adaptation and model co-adaptation. In: European conference on object-oriented programming, Springer, pp 600–624
Zaytsev V (2014) Negotiated grammar evolution. J Object Technol 13(3):1–1
Zhang W (2023) Towards automated support for the co-evolution of meta-models and grammars. Licentiate thesis, University of Gothenburg, Sweden
Zhang W, Strüber D (2024) Tales from 1002 repositories: Development and evolution of xtext-based dsls on github. In: SEAA'24: Euromicro Conference Series on Software Engineering and Advanced Applications
Zhang W, Hebig R, Strüber D, Steghöfer JP (2023a) Automated extraction of grammar optimization rule configurations for metamodel-grammar co-evolution. In: SLE, pp 84–96
Zhang W, Steghöfer JP, Hebig R, Strüber D (2023b) A rapid prototyping language workbench for textual DSLs based on Xtext: Vision and progress. arXiv:2309.04347
Zhang W, Holtmann J, Strüber D, Hebig R, Steghöfer JP (2024) Supporting meta-model-based language evolution and rapid prototyping with automated grammar transformation. JSS 214
Zhang W, Strüber D, Hebig R (2025) Dataset for 'Development and evolution of Xtext-based DSLs on GitHub: An empirical investigation'. https://osf.io/n5kfr/

Weixing Zhang became a postdoctoral researcher in the SDQ group at the Karlsruhe Institute of Technology in October 2025. He received his master's degree from Beijing Jiaotong University in 2013 and received his Ph.D. degree from the University of Gothenburg in October 2025. His research fields inlude Model-Driven Development and Domain-Specific Languages, empirical software engineering, and AI4SE. He once worked as a software engineer in the industry for seven years and has extensive software development skills.

Daniel Strüber is an associate professor at Chalmers and University at Gothenburg, Sweden, and an assistant professor at Radboud University in Nijmegen, the Netherlands. His research interests are in model-driven engineering, AI engineering, and variant-rich systems. He was awarded his doctoral degree from Philipps University Marburg, Germany, and worked as a post-doctoral researcher at University of Koblenz and Landau, Germany, and Gothenburg University, Sweden. He is a co-author of over 100 papers with six Best Paper Awards. He has been a Program Committee member of several leading conferences, including ICSE, ASE, MODELS, and a Program Chair of premier community venues, such as SPLC and ICGT.

Regina Hebig is Professor for Software Engineering at the University of Rostock, Germany. Her research interests include software evolution, AI4SE, software modelling, and software processes. She did her PhD at the University of Potsdam, Germany, in 2014 and her docent degree at the University of Gothenburg in 2019, where she worked as an Associate Professor until her change to Rostock in 2023.

Authors and Affiliations

Weixing Zhang[1] · Daniel Strüber[1,2] · Regina Hebig[3]

✉ Weixing Zhang
weixing.zhang@gu.se

Daniel Strüber
danstru@chalmers.se

Regina Hebig
regina.hebig@uni-rostock.de

[1] Chalmers | University of Gothenburg, Gothenburg, Sweden

[2] Radboud University, Nijmegen, Netherlands

[3] University of Rostock, Rostock, Germany