# Editing Support for Software Languages: Implementation Practices in Language Server Protocols

Djonathan Barros
PPGComp, Western Paraná State University
Brazil

Sven Peldszus
Ruhr-University Bochum
Germany

Wesley K. G. Assunção
Johannes Kepler University
Austria

Thorsten Berger
Ruhr-University Bochum and Chalmers | University of Gothenburg
Germany and Sweden

## ABSTRACT

Effectively using software languages, be it programming or domain-specific languages, requires effective editing support. Modern IDEs, modeling tools, and code editors typically provide sophisticated support to create, comprehend, or modify instances—programs or models—of particular languages. Unfortunately, building such editing support is challenging. While the engineering of languages is well understood and supported by modern model-driven techniques, there is a lack of engineering principles and best practices for realizing their editing support. Especially domain-specific languages—often created by smaller organizations or individual developers, sometimes even for single projects—would benefit from better methods and tools to create proper editing support.

We study practices for implementing editing support in 30 so-called language servers—implementations of the language server protocol (LSP). The latter is a recent de facto standard to realize editing support for languages, separated from the editing tools (e.g., IDEs or modeling tools), enhancing the reusability and quality of the editing support. Witnessing the LSP's popularity—a whopping 121 language servers are in existence today—we take this opportunity to analyze the implementations of 30 language servers, some of which support multiple languages. We identify concerns that developers need to take into account when developing editing support, and we synthesize implementation practices to address them, based on a systematic analysis of the servers' source code. We hope that our results shed light on an important technology for software language engineering, that facilitates language-oriented programming and systems development, including model-driven engineering.

## CCS CONCEPTS

• **Software and its engineering → Language features**; **Formal language definitions**; **Context specific languages**.

## KEYWORDS

Language engineering, code assistance, source code editor, implementation practices.


*MODELS '22, October 23–28, 2022, Montreal, QC, Canada*

ACM ISBN 978-1-4503-9466-6/22/10.
https://doi.org/10.1145/3550355.3552452

**ACM Reference Format:**
Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. 2022. Editing Support for Software Languages: Implementation Practices in Language Server Protocols. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3550355.3552452

## 1 INTRODUCTION

Software languages are paramount—not only to software engineering, but also to many other engineering disciplines that need to create models and automate tasks. Effectively using software languages—including programming languages, as well as domain-specific or general-purpose modeling languages [11]—requires effective editing support. Modern IDEs and modeling tools often come with sophisticated editing support to create, comprehend, and modify programs or models expressed in a certain language. Typical editing support features are code completion, syntax highlighting, error marking, formatting, and refactoring, among others.

Unfortunately, creating proper editing support for languages is difficult. While for mainstream software languages, the vendors of software engineering or modeling tools typically invest the necessary resources to realize editing support, domain-specific languages (DSLs) are often created by smaller organizations or individual developers. Sometimes, their use is limited to specific purposes or only a few projects. At the same time, such languages play a major role in increasing automation in software engineering [7, 64], and offering concise and semantically rich notations [9, 13]. As such, DSLs would especially benefit from better support to realize editing support—allowing their users focus on solving real problems, instead of wasting time with learning the exact use of individual DSLs.

Researchers and practitioners have worked intensively on methods and tools to build languages. Thanks to modern techniques, such as meta-modeling, automated mapping of abstract-to-concrete syntax, or model transformations, often integrated in modern and convenient language workbenches [21], the architectures of language infrastructures [30, 61] and the language engineering processes are well-understood [23, 64]. In addition to the technology on language engineering, there are implementation patterns for programming languages [39], for instance, on how to build abstract syntax trees (ASTs), on how to realize the tree walking and rewriting, and on how to realize tree pattern matching [5]. For DSLs, patterns for domain analysis, design, and implementation have been presented [33, 54]. Unfortunately, there is a lack of engineering principles and best practices for realizing the editing support of languages.

The language server protocol (LSP) [27, 35] is a recent initiative from 2016 to modularize editing support into the so-called language

servers. The LSP addresses the problem that language-specific editing support is currently deeply integrated into single IDEs or editors, preventing its reuse or extension for different tools. The LSP aims at enabling language engineers to make their languages, be it programming languages or DSLs, available to a wide range of editing tools while requiring minimal effort for adoption and reuse. The LSP describes a common API that can be implemented and reused by different clients (i.e., editors and IDEs) [35]. The tools connect to a language server, where individual editing support *features* can be requested for a program or model. The LSP calls these instances *documents*, which is the term we use in the remainder. Originally, LSP was proposed by Microsoft to provide core features for the IDE Visual Studio Code [35]. So far, a total of 23 features are defined in the protocol. It quickly became popular as a protocol used by different editors and languages [35]. As of now, 121 different language servers are listed on a community-driven curated list of LSPs [53].

We took this opportunity and investigated the design and realization of real language servers. Our goal was to identify implementation *concerns* related to the realization of language editing support. By analyzing a substantial sample of 30 LSP server implementations, we identified relevant concerns and the different realizations. We synthesized both into a set of practices on how to realize editing support that can be used by language engineers and researchers. Our working hypothesis was that engineering principles can be identified within existing language servers, paving the way to our long-term goal of establishing patterns and pattern catalogs for realizing language editing support. Among our sample, we analyzed the source code of the seven most popular LSP features. We followed a thematic synthesis method [12] by coding, analyzing, grouping, and reporting how editing features are implemented.

We hope to provide researchers with concerns related to the realization of editing support and insights on implementation practices for addressing them, to spark a discussion on best practices and eventually developing a theory and novel techniques on realizing effective editing support for languages. We hope that practitioners can use our concerns and discussed solutions as guidelines when implementing new servers or extending existing ones.

## 2 BACKGROUND

Language editing support assists developers when writing, comprehending, and changing documents (i.e., programs or models). Unfortunately, effective support needs to be language-specific, and realizing it can be costly and error-prone. Typical editing support, such as renaming, code navigation, or hover documentation, requires proper parsing and traversal of the source tree and knowing the specifics of each language [27]. Necessary engineering activities such as choosing a suitable parser, knowing the keywords, collecting runtime information, and formatting the source code properly, are not trivial. Even worse, users may prefer different editing tools, such as different IDEs and modeling tools, requiring editing support to be implemented multiple times for individual languages [50]. To this end, the API provided by the LSP, which can be implemented once and used by multiple editing tools, reduces redundancies and maintenance efforts for editing support implementations.

**LSP Overview**. The LSP [35] is a client-server protocol that enables the clients (i.e., the editing tools) to request editing support from *language servers*. The features that can be implemented are defined by the LSP specification [35]. Each LSP server can implement different editing support features depending on the respective language's characteristics or the desired editing features.

The actual editing support is encapsulated as individual methods representing a total of 23 editing support features currently specified by the LSP. A language server has, like its clients, access to the workspace that contains the documents for which editing support is required. Another abstraction, called *symbols* in the LSP, refers to any language element for which some editing support can be provided, such as identifiers, expressions, statements, or other structural concepts (e.g., classes, methods, constants).

**LSP Features**. Based on the feature descriptions presented in the LSP specification [35], we can classify the features into three different categories. Note that we list and explain concrete features and the category they belong to shortly, in Sec. 4.1, specifically Fig. 2.

*Code assistance:* The LSP offers seven features that assist developers when writing new code. Those features usually suggest computations of changes, such as renaming symbols, formatting the code, or suggesting code completions for a given prefix.

*Code comprehension:* The LSP offers 14 features to assist developers when exploring documents (e.g., source code). Those features usually do not change documents, but help developers by providing documentation and supporting the navigation through them.

*Auxiliary:* The LSP offers two features that allow LSP servers to provide additional capabilities not further specified by the protocol. One example is *Execute Command*, allowing the server to provide additional capabilities, such as refactorings.

**LSP Workflow**. To give a general understanding on the LSP implementations, we describe the typical internal workflow of how feature requests are executed, as proposed in the LSP [35]. Every feature request is independent of others and consists of four parts:

*Action identification:* When requesting a feature, the *Client* identifies the action that must be executed (e.g., *textDocument/completion*) and the execution's target document. To describe this target, the client sends a document identifier (*DocumentID*) and the *cursor position* (line and column) within the document.

*Load Document:* Based on the *DocumentID*, the server loads the target document. To do this, not only the client on which the document is being edited needs access to the documents but also the server.

*Provide action:* With access to the document, the server executes the feature, e.g., identifying what symbol needs to be completed and collecting the completion information. Some features, such as `Completion` or `Rename`, require the server to explore all documents.

*Feature response:* Finally, the server returns a response message. For instance, for the feature `Completion`, after collecting suggestions that can be inserted at the target position, the server responds with a set of *Completion Items*. Each item is a *Text Edit* operation that can be applied to perform the suggested completion on the document.

## 3 METHODOLOGY

Our study focuses on implementation practices for editing support in LSP servers. We aim to identify relevant concerns, which are aspects that should be considered when developing editing support,

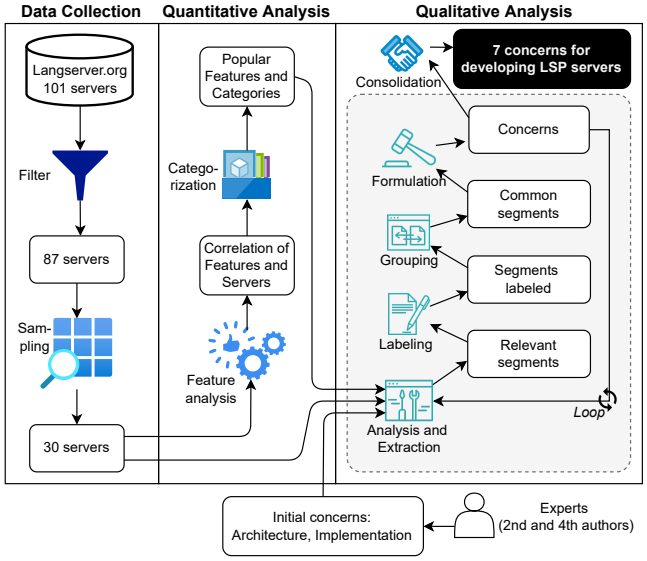


**Figure 1: Methodology overview**

and best practices addressing these concerns. Fig. 1 illustrates our methodology. First, a *quantitative analysis* illustrated in the center of Fig. 1 and described in Sec. 3.2, and second, a *qualitative analysis* illustrated on the right of Fig. 1 and described in Sec. 3.3. Further details are provided in our replication package [3].

## 3.1 Language Server Selection

As subjects for our study, we selected a sample of 30 LSP servers from a community-curated repository that collects known LSP servers [53]. While currently amounting to 121 LSPs (May 2022), at the time of the sample selection it had 101 LSPs (March 2021). We selected our sample as follows:

**No Source Code**. First, we filtered out all LSP servers for which we could not find their source code. This affected three servers.

**Not Actively Developed**. We removed 11 LSP servers listed as archived or deprecated, taking into account the information on the community repository and in the server's source code repositories (README files and "deprecated" flag). This step assured that we focus on the current state-of-practice in LSP development.

**Selection of all Java-based Servers**. We selected all LSP servers implemented in Java for the following three reasons. First, substantial language engineering tooling is implemented in Java, allowing us to compare the LSP servers with our own experiences in building DSLs in Java-based tooling (specifically, EMF, the parser generator ANTLR, and other related frameworks for language analysis and program or model transformation) [6, 14, 29, 32, 42, 43, 48, 51, 57, 63, 64]. Second, Java is among the most popular and well-understood programming languages, easing our analysis (as opposed to analyzing servers written in R, Go, Erlang, or Elixir, for instance). Third, Java is, together with Typescript, the most popular implementation language among all LSP servers documented [53].

**Random Selection among all other Servers**. Being confident about understanding the necessary details about our Java-based LSP servers, we continued the analysis for a more diverse sample.

**Table 1: Overview over the analyzed LSPs**

| | target lang. | type | impl. lang. | source repository |
|---|---|---|---|---|
| 1 | ActionScript3 | GPL | Java | github.com/BowlerHatLLC/vscode-nextgenas/tree/master/language-server |
| 2 | R | GPL | R | github.com/REditorSupport/languageserver |
| 3 | Go | GPL | Go | github.com/golang/tools/tree/master/gopls |
| 4 | Erlang | GPL | Erlang | github.com/erlang-ls/erlang_ls |
| 5 | Flux | DSL | Rust | github.com/influxdata/flux-lsp |
| 6 | Assembler | GPL | C++ | github.com/eclipse/che-che4z-lsp-for-hlasm |
| 7 | XML | DSL | Java | github.com/angelozerr/lsp4xml |
| 8 | Turtle | DSL | Type-script | github.com/stardog-union/stardog-language-servers/tree/master/packages/turtle-language-server |
| 9 | C/C++ | GPL | C++ | github.com/MaskRay/ccls |
| 10 | Java | GPL | Java | github.com/eclipse/eclipse.jdt.ls |
| 11 | Robot Framework | DSL | Python | github.com/robocorp/robotframework-lsp |
| 12 | Rust | GPL | Rust | github.com/rust-analyzer/rust-analyzer |
| 13 | Xtext (any lang.) | DSL | Java | github.com/eclipse/xtext-core |
| 14 | Python | GPL | Python | github.com/palantir/python-language-server |
| 15 | Cobol | GPL | Java | github.com/eclipse/che-che4z-lsp-for-cobol |
| 16 | Elixir | GPL | Elixir | github.com/elixir-lsp/elixir-ls |
| 17 | Puppet | DSL | Ruby | github.com/lingua-pupuli/puppet-editor-services |
| 18 | PHP | GPL | PHP | github.com/felixfbecker/php-language-server |
| 19 | Groovy | GPL | Java | github.com/prominic/groovy-language-server |
| 20 | LaTeX | DSL | Rust | github.com/efoerster/texlab |
| 21 | Apache Camel | DSL | Java | github.com/camel-tooling/camel-language-server |
| 22 | Ballerina | GPL | Java | github.com/ballerina-platform/ballerina-lang/tree/master/language-server |
| 23 | Java | GPL | Java | github.com/georgewfraser/vscode-javac |
| 24 | SonarLint | DSL | Java | github.com/SonarSource/sonarlint-language-server |
| 25 | Lua | GPL | Java | github.com/EmmyLua/EmmyLua-LanguageServer |
| 26 | MOCA | DSL | Java | github.com/mrglassdanny/moca-language-server |
| 27 | OCaml | GPL | OCaml | github.com/ocaml/ocaml-lsp |
| 28 | TTCN-3 | DSL | Go | github.com/nokia/ntt |
| 29 | Swift & C-family | GPL | Swift | github.com/apple/sourcekit-lsp |
| 30 | Vala | GPL | Vala | gitlab.gnome.org/esodan/gvls |

We randomly selected additional servers until we reached a total sample size of 30 servers, including 18 additional LSP servers.

The purpose of this sampling process was to control the selection of the LSP servers with the aim of enabling proper exploration among existing servers. Table 1 provides an overview of the studied LSP servers. We selected 30 LSPs (≈30% of the curated list), implemented in 14 and targeting 31 programming languages or DSLs. Taking saturation [25] as a quality criterion, we reached saturation after analyzing the 18th LSP server of our sample that is when we did not find any new insight (explained shortly, in Sec. 3.3).

## 3.2 Quantitative Analysis

We quantitatively analyzed the LSP servers to identify frequently implemented features and correlations among them. We focus on frequently implemented features as they provide us with a homogeneous range of implementations to be compared and help us to identify relevant concerns and common implementation practices.

To collect the features implemented in the servers, we took advantage of the information available by default in each LSP (i.e., following the LSP specification). Every LSP server has a method called *initialize* [35], which is the first method requested when a client starts using the server. This method returns an object implementing the interface *InitializeResult*, having the property *capabilities* that describes all features available in the server. Since many servers were hard to build and run, while nearly all servers had

the required information hard-coded, we statically analyzed their implementations. Thus, for every server, we inspected the source code of the *initialize* method to collect the features provided. We also obtained the versions of the LSP specification implemented. At the time of analysis, the version of the LSP specification was *3.16.0*, released on 12/2020. As it is a recent version and many of the LSPs in our sample did not implemented it at the time of analysis, we just considered features defined until version *3.15.0*, from 01/2020.

## 3.3 Qualitative Analysis

To identify frequently used implementation practices, we applied an iterative analysis approach in which we first identified relevant *concerns* [2] and related *practices* afterwards. To be more precise, as shown in Fig. 1, we proceeded as follows.

Based on the expert knowledge of the second and fourth authors, we started with two initial *concerns* focusing on the architecture and implementation-details of LSP servers. Driven by these two concerns, we systematically collected common implementation aspects for all LSP servers. First, we iteratively analyzed the individual LSP server implementations and extracted *relevant segments* (i.e., portions) of the source code that are related to the initial concerns. Thereafter, we *labeled* each relevant code statement with how exactly it addresses the initial concerns. For deriving more detailed concerns, we grouped the labeled statements based on their *commonalities*. Then, we refined these groups based on new or already identified *concerns*. We continued this process for the next LSP server until we analyzed our entire sample.

To formulate the practices addressing the identified concerns, we conducted an exploratory analysis [15], starting from the identified concerns. To this end, we analyzed the overall project structure of each LSP server and identified entry points for each considered LSP feature. After the entry point identification, we started an extensive analysis of the source code to identify implementation practices following a process inspired by thematic analysis [12]. We followed the feature implementation line-by-line but also dependencies, e.g., method calls. We captured relevant practices on a higher level as coding themes and described them as *possible values*, which served as the codes of the thematic analysis. Whenever we saw an interesting aspect of the implementation (libraries, code patterns, interaction with other features/resources), we added a new code. Then, we consolidated these codes in a group discussion to formulate practices related to the concerns.

# 4 CONCERNS AND PRACTICES

We identified concerns by looking into typical software implementation aspects and considering activities such as software design (i.e., architecture, including ways, how the system is structured), followed by implementation practices for realizing the planned LSP server. All concerns are specific to language engineering, e.g., we looked at how instances are represented in a server (i.e., as an AST or a simple string of characters on which the editing support is implemented). In what follows, we present, discuss, and answer each identified concern with the practices observed in our study. The first concern is based on our quantitative analysis, while the remaining six concerns emerged and were answered during our qualitative analysis.

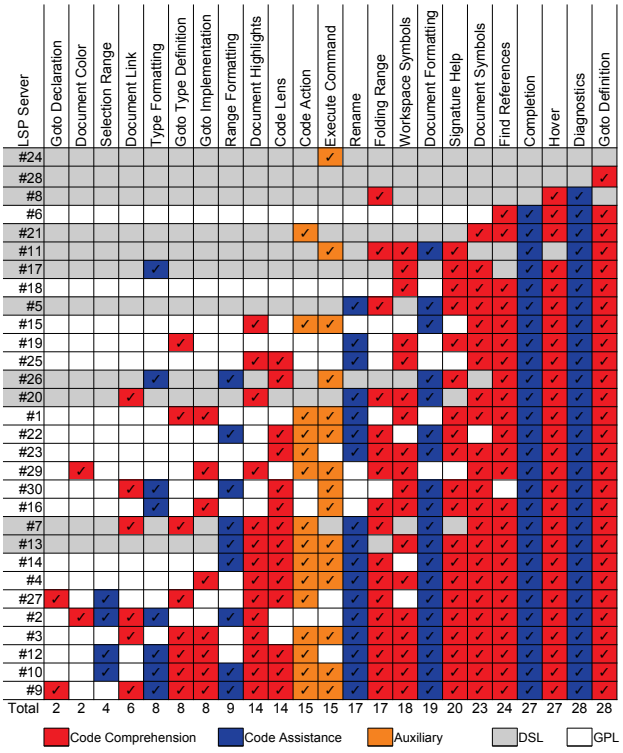


**Figure 2: Features implemented in each LSP server**

## 4.1 Concern: Selection of Editing Features

When implementing a new LSP server, it is useful to know which features are typically provided by LSP servers. To get a general overview of possible features supported by LSP servers, we quantified all features provided by the LSP servers in our sample.

*4.1.1 What are the most frequently implemented features?* Figure 2 shows which features are implemented by which server. LSP features are in the columns, and servers are in the rows. Both are sorted in ascending order by the number of implemented features or servers implementing the feature, respectively. We discuss the provided features according to what kind of editing support they offer and their frequency in the LSP server implementations. Understanding the frequency of the individual editing features can help engineers in deciding which features to implement first, as well as researchers in focusing their research on industry needs.

`Goto Definition` and `Diagnostic` are the most frequently implemented features. These features constitute common editing support for languages [27]. Only two servers miss these features, servers #24 and #8 for `Goto Definition` and servers #24 and #28 for `Diagnostic`. These servers are special, they use LSP as a vehicle to offer third-part services, such as offering a linter (#24).

Due to space limitations, we cannot describe all features in detail and refer to the LSP specification [35]. Next, we focus on the most frequently implemented third of LSP features, totaling seven features. Reducing the number of features enabled us to analyze their implementations in-depth. In what follows, we summarize these seven features and the servers implementing them.

(1) `Goto Definition` (*Code Comprehension*, 28 servers) allows users to quickly navigate within a project to find where a reference (e.g., a variable) is defined.
(2) `Diagnostic` (*Code Assistance*, 28 servers) returns results/errors from compilers, parsers or even lint tools.
(3) `Hover` (*Code Comprehension*, 27 servers) shows information related to a given text document position.
(4) `Completion` (*Code Assistance*, 27 servers) computes a list of completion items at a given cursor position.
(5) `Find References` (*Code Comprehension*, 24 servers) collects all references pointing to the symbol at the target position.
(6) `Document Symbols` (*Code Comprehension*, 23 servers) returns either all symbols present in a document or even the entire hierarchy of the present symbols[35].
(7) `Signature Help` (*Code Comprehension*, 20 servers) provides additional information such as what is the parameter currently selected or the method's documentation.

When investigating the seven most frequent features, we noticed that these are the more generically specified features of the list in Fig. 2, which give great flexibility on how and to what extent to implement them to the developers. Altogether, these features cover different aspects of language support ranging from resolving references (`Goto Definition` and `Find References`) over aggregating relevant information (`Hover`, `Document Symbols`, and `Signature Help`) to features providing more complicated tasks (`Diagnostic` and `Completion`). This variety indicates that editing support depends not primarily on a single kind of editing feature, but needs rather diverse features.

Three LSP servers of our sample implement noticeably few features (#8, #24, #28) and use LSP, as mentioned above, just as a basis to offer third-party services. *SonarLint* (#24) only implements `Execute Command` to forward the SonarLint [52] standalone-tool's static code analysis results. Similarly, *nokia/ntt* (#28) is an LSP server for language-agnostic testing with TTCN-3, a DSL for scripting tests. Finally, *Turtle* (#8), which is the basis of the Stardog IDE [55], only implements `Folding Range`, `Hover`, and `Diagnostics` as a wrapper for a data exploration tool. Altogether, LSP servers implementing only a few features usually use LSP to forward the functionality of powerful third-party libraries.

*4.1.2 Which features to implement for a language?* Previously, we investigated what are the most frequently implemented editing features and discussed possible reasons. Despite knowing what can be provided by LSP servers, due to limited resources, this is not enough information to develop an LSP server. The challenge is to decide which features to implement. Thereby, we can have two influencing factors. First, language-specific properties, such as language paradigms, can impact the feasibility of the different features. Second, there can be an interaction among the features, e.g., features complementing each other.

**Relations Between Languages and LSP Features**. For all servers, we calculated the correlation between implemented features and language characteristics of the target languages. We used the R implementation [49] of the Spearman's rank correlation [41]. Figure 3 shows the resulting correlation matrix with the language characteristics on the vertical axis and the features as well as the number of

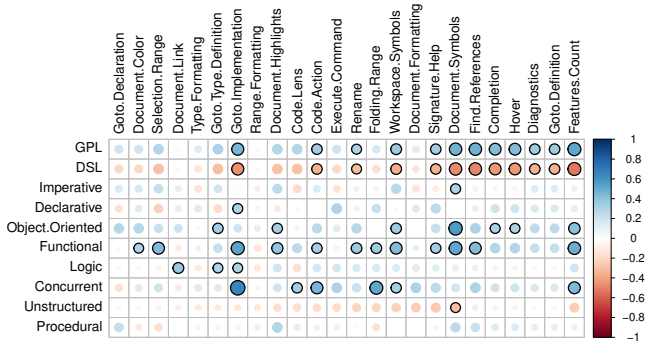


**Figure 3: Correlations between editing features and paradigms of instance languages (bold circles are significant at a significance level of 5%).**

implemented features on the horizontal axis. As language characteristics, we consider language type (general purpose language (GPL) and DSL) and language paradigms (e.g., imperative, declarative, or functional). Based on our sample size, all correlations with an absolute correlation higher than 0.306 are statistically significant at a level of 5% [31]. Altogether, we observed 55 significant correlations of which we discuss the most interesting ones in what follows.

First, GPLs tend to have a positive correlation with all LSP features whereas DSLs tend towards negative correlations, meaning that servers for GPLs are in general likely to implement features while servers for DSLs are likely to not implement features. We can also see this correlation in Fig. 2, where DSLs are shown rather on the top. However, popular DSLs such as XML, LaTeX, or Xtext still implement many features. Therefore, the number of implemented features could mainly depend on the language's popularity. This is supported by the popular languages Java, C/C++, and Rust that are used to implement the most LSP features.

When looking at the individual correlations, we notice that there is a strong correlation with *Functional* and *Concurrent* languages for the feature `Goto Implementation`. Practically, there is a distinguishment between signature and implementation in such languages, explaining the observed popularity. While one would also expect any *Object-Oriented* language to be supported by this feature, this is only easily possible for class-based languages but not for prototype-based languages such as Python.

*Object-Oriented* and *Functional* languages have a rather strong correlation with `Document Symbols`, reflecting the strong structuring in such languages and the need to assist developers by providing all language constructs contained in a file.

For all other LSP features we cannot find strong correlations in our example, either because they are implemented by nearly all LSP servers or they are implemented only in a few cases, making them more specific to individual considerations. In summary, besides features that should be implemented in nearly all LSP servers, there is some editing support specific to language paradigms. Still, the popularity of a language seems to be the main driver in implementing editing support and might also provide the resources to implement less relevant features.

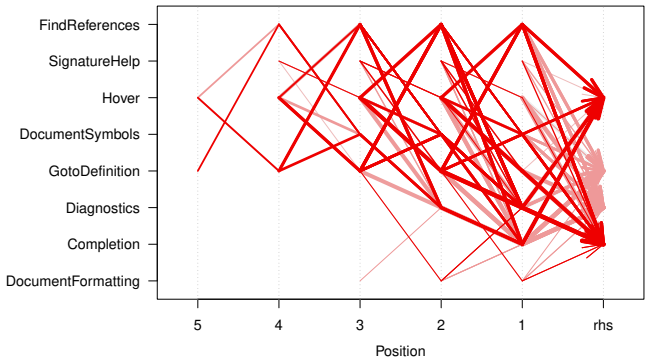


**Figure 4: Parallel coordinates plot for 167 rules**

**Relations Among LSP Features**. To identify features usually implemented together, we utilized association rule mining that discovers associations among data items [1], like if *A* and *B* are implemented, *C* is implemented most likely, too. For that, we used the algorithm Apriori, available on the R package *arules* [59]. As parameters, we set the thresholds *support* to only consider features in at least 63% of the servers, and *confidence* to 0.95, which means that the features must appear together in at least 95% of the cases.

The algorithm mined 167 associations rules, summarized in Fig. 4. Positions 1 to 5 indicate how many features are in the *antecedent*, and rhs indicates *consequent*. The thickness of the arrows represent the *support*, namely how strong is the association—thicker means stronger, and the shading the *lift*, which indicates how likely is the association to happen—darker means more likely.

By analyzing the data, we can observe: (i) there are few rules with five or four features, mostly involving `Goto Definition` and `Hover`, which are the first and the third most frequent features; (ii) there are no rules in which the *consequent* are the features `Find References`, `Signature Help`, `Document Symbols`, and `Document Formatting`; (iii) the rules with stronger *lift* have as *consequent* `Hover` and `Completion`; (iv) `Find References`, `Diagnostic`, and `Completion` are among the features with stronger *support*. Overall, these results show that when servers have the popular features `Find References`, `Diagnostic`, and `Goto Definition`, they will most likely also have `Hover` and `Completion`.

## 4.2 Concern: Use of Third-Party Libraries

Libraries are a common practice for sharing and reusing functionalities [26]. In some cases, even the entire language support can be realized by wrapping libraries. Therefore, we are interested in the roles of external libraries in the LSP implementations and searched for recurring kinds of libraries.

*4.2.1 Commonly Used Libraries.* For many recurring problems, libraries provide mature solutions. We identified six different kinds of libraries that are frequently used in the LSP server implementations.

**Compiler/Parser (25 servers)**. Most frequently, LSP servers use compilers or parsers to create a document's abstract representation, which they traverse to collect required information, e.g., reference positions or suggested refactorings, such as renaming a symbol. One example is the library Tolerant PHP Parser [34].

**Parser Generators and Maintaining a Grammar (3 servers)**. Comparable to compilers or parsers, custom grammars allow the creation of an abstract syntax representation from a document's plain contents, but the grammar that describes the tokens of the target language is part of the LSP project and used to generate a parser (e.g., using Antlr [40]). Besides the LSP features, the LSP server has to maintain the grammar, e.g., when language constructs are introduced or deprecated as part of the target language's evolution. For instance, the server *MOCA* (#26) maintains a grammar for the DSL MOCA (see Table 1).

**Documentation Handler (5 servers)**. Documentation such as JavaDoc is usually declared in another syntax as the target instance. It is ignored by parsers and can even be stored separate documents. Libraries such as Jedi [28] for Python, ElixirSense [20] for Elixir, and DocBlock [45] for PHP can handle such documentation and map it to corresponding instance elements.

**Environment Info Handler (9 servers)**. Besides information from the documents and the language specification, environment-specific information (e.g., libraries or the installed language environment) can be beneficial for features such as `Code Completion`. For instance, the libraries M2Eclipse [19] and Buildship [16] can collect information on Maven and Gradle dependencies of Java projects.

**Linter (3 servers)**. The LSP feature *Diagnostics* provides information such as wrong syntax. In addition to compiler reports, linters and analysis tools are used to collect diagnostics. For instance, the library pytlint [47] is used to validate the files and return warnings related to source code formatting, the presence of smells, or recommended refactorings (server #14). The server *Elixir* (#16) uses the static analysis tool Dialyzer [22] to provide similar diagnostics.

**LSP SDK (13 servers)**. There are a few SDKs helping engineers to deal with LSP-related details (e.g., schemata of messages). We found usages of LSP4J [18] for server implementations written in Java as well as *Microsoft/vscode-languageserver-node* [36] targeting servers implemented in Javascript/Typescript and running in Node.js.

In summary, libraries are mainly used to avoid custom implementations for accessing the document instances or additional needed information for realizing LSP features.

*4.2.2 Wrapped Language Support Libraries.* While libraries mainly support LSP feature implementations, 7 servers benefit from libraries that implement LSP features entirely or partially. For example, Eclipse JDT [17] is a library used by the *Java* server (#10) that implements features including `Completion` and `Find References`. We observed the same with the library Jedi [28] used by the *Python* server (#14). Those libraries provide enough features so that the server becomes a wrapper around them, which only needs to pass the document to the library and wrap its response, respectively.

Considering the decreased effort when utilizing libraries to implement any aspect of an LSP server, engineers still have to consider extra efforts to translate interfaces and wrap responses to utilize library (e.g., translating linter diagnostic messages). Defining interfaces to encapsulate and avoid the libraries' interfaces polluting the server interfaces, potentially using design patterns (e.g., *Adapter* or *Facade* [24]), must also be considered by LSP developers.

## 4.3 Concern: Structuring LSP Servers

Architecture plays an essential role in software engineering and is crucial for the successful development and maintenance of software systems [4, 24]. In the literature, various architectural patterns and

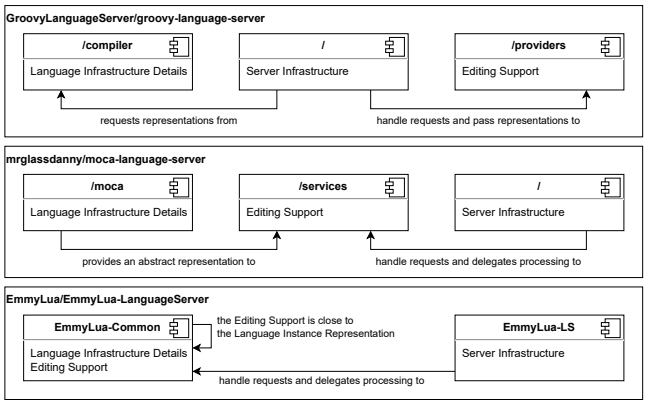


**Figure 5: Example of server layers observed**

principles [24, 58] have been proposed, but when incorrectly applied, anti-patterns [8, 44] can emerge that can challenge realizing a software system. Accordingly, we have to identify suitable design principles for LSP servers.

When investigating the source code of the LSP servers, we identified two design principles that were applied or are applicable to all investigated LSP server implementations. First, the architecture of LSP servers can be realized based on a layered architectural style. Second, LSP servers must consider the overall flow to completely implement a feature when deciding how to structure the internal communication of the components.

*4.3.1 Layered Architecture:* We observed that LSP servers are mainly structured into three layers (*Server Infrastructure*, *Language Infrastructure*, and *Editing Support*) with different responsibilities for the LSP feature execution.

Figure 5 shows examples of the layered structure of LSP servers. The server *Moca* (#26) is structured into well-defined package. The package *moca* contains code to deal with the *Language Infrastructure* details, such as parsing and creating an abstract instance representation, the package *services* the actual implementation of *Editing Support* capabilities, and the project root contains code to deal with the *Server Infrastructure*. A similar packaging structure is followed by *Groovy* (#19) and *Latex* (#20).

However, not every server follows this structure. For example, *Lua* (#25) has two main components: (i) *EmmyLua-Common* that aggregates the *Language Infrastructure* and *Editing Support*, and (ii) *EmmyLua-LS* providing *Server Infrastructure*.

Altogether, the responsibilities of the layers are as follows:

**Language Infrastructure**. This is the lowest layer and implements the immediate interaction with the documents (e.g., parsing the source code for creating an abstract syntax representation). If needed, support for multiple languages can be provided within this layer. The input for this layer is usually a document and the output is an instance representation that can easily be processed to realize LSP features, such as an abstract syntax tree (AST). As the parsing process can generate additional relevant information (e.g., warnings/errors observed at parsing or compilation), it can also implement patterns such as *Publish/Subscribe* to provide such information to the *Editing Support* layer above it.

**Editing Support**. This layer realizes the feature-specific processing. For example, traversing the abstract representation to collect completion items. Some servers opted to delegate this layer's responsibility to language support library that provide features including *Rename* and *Completion* (cf., Sec. 4.2).

**Server Infrastructure**. This layer is responsible for dealing with protocol details (e.g., handling feature requests). Some protocol details can be delegated to an LSP SDK. A curated list of LSP SDKs for different languages can be found at the official LSP website,[1] and we discuss observed examples in Sec. 4.2.

Even when the servers encapsulate code in the same layers, the information flow might differ. Therefore, in what follows, we discuss the most commonly observed execution flow.

*4.3.2 Feature Execution Flow:* For the servers not only wrapping libraries, we identified the feature execution flow described in Sec. 2. After receiving a request, the server loads the document into memory and then calls for each feature a provider that handles it. The feature execution is usually implemented in three steps:

**Parse**. Navigating through a document's symbols is usually needed to implement features such as *Completion* and *Find References*. To this end, most of the servers analyzed need to parse the document into an abstract instance representation in the first execution step.

**Traverse**. The abstract syntax representation is then traversed to collect the elements that will be needed to process the feature (e.g., all AST nodes that are symbols in a specific document).

**Process**. The collected elements are then processed to realize the respective LSP feature. This can involve filtering, ordering, or collecting additional information that is required to process the feature.

Besides this being the most common sequence, specific cases where the execution of a feature is delegated to a library may result in a different sequence of steps. One example is the feature *Diagnostic*, which usually returns results or errors collected from a compiler, parser, or even a linter library. The LSP server just translates the messages into the response format required by the client, making all other execution steps within the LSP server obsolete.

The observed flow is predestinated to applying the Pipes and Filter pattern. However, while we found indications of this pattern, no LSP server strictly follows the pattern. Altogether, LSP server implementations can be effectively structured into layers containing task-specific filters that can be flexibly exchanged (e.g., to support multiple instance languages).

## 4.4 Concern: Implementation Granularity

When implementing an LSP server, one has to decide how granular the implementation of features should be, and whether it makes sense to separate feature implementations (and for which features) into separate modules. We investigated every LSP server implementation and assigned a granularity level based on a 5-level Likert scale ranging from the most coarse-grained (1) to more fine-grained (5) granularity of the implementation:

**1: Single Monolith (1 server)**. This granularity describes servers that implement all provided features in a single structuring entity (e.g., only one class). Just *SonarLint* (#24), which does not implement any feature on its own, is implemented at this granularity level.

[1] https://microsoft.github.io/language-server-protocol/implementors/sdks/

**2: Delegator per LSP Feature (6 servers)**. At this granularity a single file that consists out of delegator methods is used to call a library that will effectively implement the feature. This was the case for almost all servers that reuse *Language Supporting Libraries*, such as the *Python* server (#14). The only server that uses *Supporting Libraries*, but that is not on this granularity is *Swift* (#29), which besides delegating the execution to external libraries, also implements support for multiple languages (level 5).
**3: File/Class per LSP Feature (13 servers)**. The most common case is where the server has one file or class per feature with encapsulated methods/functions to implement details of the features (e.g., traversal strategies or even validation of the context where a symbol applies).
**4: Module of Multiple Files/Classes per LSP Feature (8 servers)**. The features' implementation details are well encapsulated in multiple files/classes as the implementation's complexity requires additional structuring. Among others, those feature implementations contain non-trivial searches for specific content or provide contexts to select AST nodes, e.g., variable/function completion in *Ballerina* (#22), or even have their own set of possible values, e.g., snippets/keywords completions in *Robot* (#11).
**5: Multiple Modules per Target Language (2 servers)**. At this granularity, each feature is implemented in multiple modules (e.g., because the server provides mechanisms to implement the features for multiple target languages). For example, *Turtle* (#8) is one of the supported languages in the server *Stardog*. The support for all its target languages is implemented as extensions of an *AbstractLanguageServer* that orchestrates the language-specific feature implementations execution. The editing support for each target language is encapsulated in a module that must provide methods to parse documents and provide diagnostics when changes happen. While the server *SourceKit* (#29) is based on a *Language Supporting Library*, it also implements the logic to delegate the feature execution to multiple language-specific libraries (e.g., the *Swift* programming language) and translates the result to the expected response format (e.g., a *HoverResponse*).

The servers tend to have straightforward implementations with one file per feature as the most common granularity. We can observe a tendency to finer-grained structuring when the features of LSP servers get more complicated, e.g., when a feature is implemented to support multiple languages.

## 4.5 Concern: Language Instance Representation

For providing editing support, it usually necessary to have an intermediary representation of the documents. While we observed that in some cases LSP servers can immediately operate on concrete syntax (e.g., source code representations), for most tasks a structured instance representation in abstract syntax is needed.

*4.5.1 Abstract Syntax Representation.* To make information about the program or model instances accessible, most LSP servers (25 servers) use ASTs. An AST is a structured, tree-based representation of the concrete syntax, in which nodes of the tree represent the symbols and the edges are the relations between the symbols. Also, simpler representations, such as the Domain Object Model (DOM) are used (e.g., in the R server (#2)), or servers immediately work on the plain documents (e.g., SonarLint (#24)).

*4.5.2 Tree Traversal.* From identifying a symbol or even the parameters of a function node to collecting all variables declared in a document, traversing the abstract representation of a document is a very common task realized in LSP feature implementations. This traversal can follow different strategies, from standard graph traversal algorithms to ad hoc implementations for specific cases to design patterns and third-party code.

As abstract syntax representations are usually instances of trees or graphs, the most common strategy (13 servers) is to traverse them using standard algorithms (e.g., breadth first search). Among others, this strategy is used to pre-process the document structure (*Go* server, #3) and create auxiliar index structures (cf. Sec. 4.6) or to collect nodes when implementing a feature such as `Find References`.

Simple ad hoc algorithms for navigation are used by 10 servers. for instance, the feature `Completion` in the *XML* server (#7) simply navigates all parent nodes to suggest completions. Another usage of such a bottom-up navigation is identifying the scope in which a symbol is declared (e.g., whether a variable is below a function node or a class definition). Finally, ad hoc traversal implementations are also used to determine if the cursor position is within some specific semantic structure, such as the right-hand side of an assignment statement (*Flux* server, #5).

To make the traversal implementation reusable for multiple features without implementing it each time, the *Visitor* pattern [24] was used in 10 servers. The possibility to implement server-specific visitors is commonly provided by compilers or language support libraries (e.g., *javac*'s *TreePathScanner* [38], used by the server #23). LSP servers use visitors to filter tree nodes or to accept nodes defined by the visitor logic when running the feature `Completion` (*C/C++* server, #9). Besides object-oriented implementations, other variations utilize lambda functions (e.g., *rust-analyzer* uses Hir [60] to collect expected results). Some libraries like *JDT* and *Clang/sema* go one step further and provide declarative ways to specify node characteristics for which the libraries directly execute features such as `Completion`, `Hover`, or `Find References`. Lastly, we found cases of using XPath for navigation, such as in the *R* server (#2).

Altogether, many libraries provide sophisticated support for representing and traversing documents for all major languages. If no suitable library is available for a target language, the needed functionality can be generated after providing a grammar of the target language (e.g., using *Xtext*). Only when using libraries is not possible, or when there are good reasons for not using a library, LSP server developers need to implement the abstract representation as well as the traversal on their own.

## 4.6 Concern: Performance

For providing efficient and usable editing support, response times are critical for the document editing experience by users. Consequently, we expected to observe optimization strategies (e.g., caching) to improve the features' execution performance. For example, avoiding constantly re-parsing documents can increase performance. We identified three common optimization strategies:
**AST Caching**. To avoid frequent recreation of abstract representations, these are stored in memory with a key-value relation that

matches the file URI and the data structure (e.g., the root node of an AST). The data structure can be loaded when the user notifies that the file was opened (like in the *XML* server, #7) or even in the server's startup phase where all files of the workspace are compiled and cached. Here, it is required to implement mechanisms for updating the abstract representation after changes to the document.

**Auxiliary Index Structure**. A strategy used by 13 servers to access a representation of the document but avoiding direct operations on the abstract representation is to provide an auxiliar index structure that contains relevant nodes of the parsed document. The strategy is to once collect relevant nodes from the abstract representation and store them in a navigable data structure. One concrete representation of such an auxiliary index structure is a symbol table. As the indexed structure stores the information about the symbols within a document, it can simplify the implementation of LSP features. Developers do not need to explicitly implement the traversal of the representation and to differentiate between the different node types for each feature implementation. In addition, it can also increase performance as information collected once can be reused for future feature calls. As with the abstract representations, this indexed structure needs to be reprocessed every time a document is changed (e.g., as implemented in the *Cobol* server, #15).

**Enclosing Scope Scanning**. In addition to using an abstract representation to implement LSP features, 22 servers also directly operate on a document for specific cases, avoiding the overhead of constructing additional data structures. In principle, simple string operations relying on navigating the text character by character can be used to scan the document and collect information (e.g., where the current symbol begins). This information can also be gathered by regular expressions, applied to either, the entire document or just the target line, like the *Elixir* server (#16) does. As such information can usually be collected from the AST node attributes there is no need for explicitly processing the plain document.

Some servers check syntactic correctness through the direct analysis of the documents by relating a specific position in a document to other positions (e.g., to identify whether a symbol is located within any brackets or not). Using such information, the feature implementation can determine if the symbol is inside the parameters section of a method declaration or a function call (*Xtext* server, #13). Other applications of this kind of logic is to identify if a special character is preceding the current symbol (*Assembler* server, #6) or to identify if the cursor is within a comment block (*R* server, #2).

Altogether, optimizations mainly focus on reducing the run-time overhead for building intermediate document representations as well as for searching these. As a downside, developers need to handle updating the cached data when document changes emerge, which is the subject of the next concern.

## 4.7 Concern: Handling Instance Changes

When providing editing support for languages, the constant change of documents is omnipresent and has to be considered to preserve the validity of the feature executions. The LSP protocol already defines that the server must be notified when changes happen or documents are saved [35]. Any performance optimization, such as in-memory abstract representations and even index structures must be reprocessed when they do not reflect the state of the related resource anymore. The LSP specification establishes the *textDocument/didChange* and *textDocument/didSave* requests to notify the LSP server that the resources were updated.

Among our servers, the most common case (19 servers) of synchronization is the incremental application of the changes to the resources, while nine servers synchronize the text based on the full content. The servers *SonarLint* (#24) and *Turtle* (#8) do not update the text content, but reload the entire content at every feature call.

The strategies to update the representations with the changed content are (i) *reprocess* the file, parsing it again and recreating the abstract representation; (ii) *invalidate* the file, marking it as stale or cleaning the in-memory representation, so the file will be reprocessed when the next feature is requested. In our sample, 13 servers reprocess upon *textDocument/didChange* requests, while seven invalidate. Upon *textDocument/didSave* requests, five servers reprocess and two servers invalidate the document.

In summary, handling document changes is essential for properly providing editing support. We identified two strategies for tracking document states, identifying the need to update caches, and actually reacting to document changes.

# 5 DISCUSSION & LESSONS LEARNED

Recall that providing good editing support requires implementing very different kinds of editing features. As such, our results on what are the most implemented editing features, together with their relations to language paradigms, can guide LSP developers when prioritizing the editing features.

Interestingly, one does not always need heavyweight and full-fledged language processing implementations. For instance, we observed that using abstractions other than ASTs for representing language instances can be sufficient. Also, auxiliary index structures can avoid manual traversals of the AST for each feature request. We observed that tree traversal is implemented repeatedly for different purposes and has a high potential for reuse. However, this might lead to more complicated server architectures than the ones that we observed in our study.

The layered architecture proposed by us can easily support such reusable mechanisms and allows flexibly connecting different implementations. Providing more sophisticated editing support, e.g., supporting multiple languages or more sophisticated analyses will result in a shift from one class per feature, as mainly observed in our study, toward multiple classes per feature. Further modularizing LSPs is still an open research problem.

In general, libraries help to implement editing support, but cannot do all the work and even generate additional concerns such as how to encapsulate the external library's interfaces to avoid the pollution of the project interfaces. In this regard, generated editing support can be one way to go for standard editing support. Most servers for DSLs (as opposed to the servers for programming languages) use ANTLR (4 servers), but only for parser generation. The editing support implementation is hand-written, similar to the other LSPs. Only the LSP realization of Xtext generically provides editing support for any Xtext-based language, making the implementation more fine-grained than the hand-written ones. It does not target a specific language, but any that can be expressed in an Xtext grammar. For instance, in a *Completion* request, possible symbols are looked up from

the grammar. In the end, the generated editing support implementation is the same as when Xtext generates an editor plugin for Eclipse.

All the languages supported by our language servers are so-called parser-based languages. A complementary paradigm is projectional editing (a.k.a., syntax-directed or structured editing) [6, 62], where a user's editing gestures directly change the underlying AST, without any parsing involved. The core benefits are basically unlimited language composition and flexible notations (textual and visual ones). While we did not observe any specific support, the fact that LSPs have been developed for graphical modeling languages, such as Eclipse GLSP,[2] shows that LSP can be used for projectional-editing-based languages, such as implemented in Jetbrains Meta Programming System (MPS). The main concern is to represent and pass the location of elements edited, which cannot be based on a line number and character number pair, but requires some other locator mechansms (e.g., node ID).

## 6 THREATS TO VALIDITY

**External Validity**. First, our focus on LSP to investigate editing support might bias our results unnecessarily towards LSP technicalities, limiting generalization to editing support in general. Still, LSP is a widely used protocol for providing tool-independent editing support, and the communication between clients and the LSP server is only a minor part of the server implementations. Second, our selection of LSP servers could have biased our findings (concerns and solutions). To mitigate this threat, we followed a four-step selection process to retrieve a suitable sample. We also cover a good range of languages, the sample of servers we studied provides support for 34 different languages. Our server sample also relies on 14 different implementation languages, reducing the bias of our findings. Third, our results could be biased towards either DSLs or programming languages. However, our sample contains 11 of the former and 19 of the latter, reducing bias.

**Internal Validity**. First, during our analysis, we might have misunderstood some LSPs' internals. However, our selection strategy assured that we first investigate Java-based servers, relying on our experience with Java-based language engineering tooling. Second, we started to look into architecture and then implementation, specifically, our starting points were the seven most frequently implemented features. This selection could miss important practices only seen in the other features. However, our goal of supporting other developers when implementing their own LSP server or "popular" features justifies this scope and potential bias. Third, we used the well-established thematic analysis method. After collecting the data of each server, the authors extensively discussed the codes, labels, and refined the concerns.

**Conclusion Validity**. Even tough, the implementation practices identified and discussed in this work give detailed insights into how to implement editing support, these could be considered as somewhat limited to the observed frequencies of the practices along the concerns analyzed in Sec. 4 and Sec. 5. Still, the work builds a solid foundation for further assessment of which practices work best for the least amount of effort, and which practices go hand in hand across the considered concerns.

[2] https://www.eclipse.org/glsp/

## 7 RELATED WORK

Rodriguez-Echeverria et al. [50] propose an LSP infrastructure for graphical modeling languages. Their goal is decoupling the editing support from the graphical language. To this end, they studied different alternatives to the LSP. In contrast, our work focus on more general practices for implementing LSP servers.

Erdweg et al. [21] present details of language workbenches for developing GPLs and DSLs. They investigate the implementation of 10 language workbenches regarding feature coverage, size, and required dependencies. However, nothing is explored concerning editing support. Additionally, their work is somewhat dated (from 2013) regarding recent advances in language engineering and editing support. Recall that LSPs appeared in 2017.

Bünder and Kuchen [10] present how the LSP can be used to satisfy both developers and domain experts, with different preferences, while working on the same projects. LSP provides the means for integrating different editors in model-driven software development projects. In a recent book [27], the developers of the Ballerina Language Server (a cloud-native programming language) describe the requirements to implement editing support using LSP. Then, they detail the implementation for the client and the server of the Ballerina language. Both works are experience reports for a specific scenario or technology stack, which is different from our goal of systematically studying implementation practices in many servers.

Stolpe et al. [56] describe the use of LSP to implement code editing that is reusable for any IDE with LSP support. However, they focused exclusively on the Truffle framework as a target language. In contrast, we describe details of implementation practices, considering several types of target languages. Mészáros et al. [37] also leverage LSP to integrate a code comprehension tool named CodeCompass as part of Visual Studio Code. With a similar purpose, Pupo et al. [46] use LSP to integrate a machine-learning-based tool for improving Javascript security into an IDE. These papers mainly describe the implementations for these specific cases without discussing concerns or practices that must be taken into account.

## 8 CONCLUSION

Effective editing support for software languages is crucial. We presented the first set of engineering practices, systematically identified—quantitatively and qualitatively— from a sample of 30 LSP servers. We synthesized seven core concerns and present practices for realizing language editing support. We found that a variety of features are required to provide editing support, and the concrete aspects of languages play a rather minor role in the basic editing support. Still, for advanced editing support, characteristics of the target language become more important. For designing LSP servers, we identified design and implementation practices that we hope to extend to a reference architecture in later works. When it comes to the concrete implementation-level realization of LSP servers and editing support in general, for many implementation aspects, mature libraries exist and are already widely used. In addition, we observed multiple frequently used optimization techniques that usually come together with the challenge of updating a caching structure. Also, to address this challenge, we identified ways to represent the language instances, to traverse these representations, and to handle changes to them, which must trigger updates of these representations.

# REFERENCES


[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast Algorithms for Mining Association Rules. In *20th International Conference on Very Large Data Bases (VLDB)*, Vol. 1215. 487–499.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.*

[3] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. 2022. *Supplementary Material – Editing Support for Software Languages: Implementation Practices in Language Server Protocols 59.* https://doi.org/10.5281/zenodo.6974153

[4] Len Bass, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice.* Addison-Wesley Professional.

[5] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. 2015. Modular Language Implementation in Rascal – Experience Report. *Science of Computer Programming* 114 (2015), 7–19. https://doi.org/10.1016/j.scico.2015.11.003 LDTA (Language Descriptions, Tools, and Applications) Tool Challenge.

[6] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE).*

[7] M. Bialy, V. Pantelic, J. Jaskolka, A. Schaap, L. Patcas, M. Lawford, and A. Wassyng. 2017. Software Engineering for Model-based Development by Domain Experts. In *Handbook of System Safety and Security*, Edward Griffor (Ed.). Syngress, 39–64.

[8] William H. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley & Sons.

[9] Hendrik Bünder. 2019. Decoupling Language and Editor-the Impact of the Language Server Protocol on Textual Domain-specific Languages.. In *MODELSWARD.* 129–140.

[10] Hendrik Bünder and Herbert Kuchen. 2020. Towards Multi-editor Support for Domain-specific Languages Utilizing the Language Server Protocol. In *Model-Driven Engineering and Software Development*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić (Eds.). Springer International Publishing, Cham, 225–245.

[11] S Cook, C Bock, P Rivett, T Rutt, E Seidewitz, B Selic, and D Tolbert. 2017. Unified Modeling Language (uml) Version 2.5. 1. *Object Management Group (OMG), Standard* 12 (2017).

[12] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *International Symposium on Empirical Software Engineering and Measurement.* 275–284.

[13] Swaib Dragule, Thorsten Berger, Claudio Menghi, and Patrizio Pelliccione. 2021. A Survey on the Design Space of End-user-oriented Languages for Specifying Robotic Missions. *Software and Systems Modeling* (Feb. 2021). https://doi.org/10.1007/s10270-020-00854-x

[14] Swaib Dragule, Thorsten Berger, Claudio Menghi, and Patrizio Pelliccione. 2021. A Survey on the Design Space of End-User Oriented Languages for Specifying Robotic Missions. *International Journal of Software and Systems Modeling* 20, 4 (2021), 1123–1158.

[15] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering.* Springer, 285–311.

[16] Eclipse Foundation. 2022. *Eclipse Buildship: Gradle integration for the Eclipse IDE.* https://github.com/eclipse/buildship Accessed: 2022-04-10.

[17] Eclipse Foundation. 2022. *Eclipse Java development tools (JDT).* https://www.eclipse.org/jdt/ Accessed: 2022-04-10.

[18] Eclipse Foundation. 2022. *Eclipse LSP4J.* https://github.com/eclipse/lsp4j Accessed: 2022-04-10.

[19] Eclipse Foundation. 2022. *M2Eclipse.* https://www.eclipse.org/m2e/ Accessed: 2022-04-10.

[20] Elixir Language Server Protocol. 2022. *ElixirSense.* https://github.com/elixir-lsp/elixir_sense Accessed: 2022-04-10.

[21] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering (SLE).*

[22] Ericsson AB. 2022. *Dialyzer User's Guide.* https://www.erlang.org/doc/man/dialyzer.html Accessed: 2022-04-10.

[23] Martin Fowler. 2010. *Domain-specific Languages.* Pearson Education.

[24] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, John Vlissides, et al. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Pearson Deutschland GmbH.

[25] B.G. Glaser. 1978. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory.* Sociology Press. https://books.google.at/books?id=73-2AAAAIAAJ

[26] Martin L Griss. 1993. Software reuse: From library to factory. *IBM systems journal* 32, 4 (1993), 548–566.

[27] Nadeeshaan Gunasinghe and Nipuna Marcus. 2022. *Language Server Protocol and Implementation.* Apress. https://doi.org/10.1007/978-1-4842-7792-8

[28] Dave Halter. 2022. *Jedi - an awesome autocompletion, static analysis and refactoring library for Python.* https://jedi.readthedocs.io/en/latest/ Accessed: 2022-04-10.

[29] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wasowski. 2018. Model Transformation Languages Under a Magnifying Glass – A Controlled Experiment with Xtend, ATL, and QVT. In *26th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE).*

[30] Ralf Lämmel. 2018. *Software Languages.* Springer.

[31] Peter M Lee. 2005. Upper Critical Values for Spearman's Rank Correlation Coefficient $R_s$. https://www.york.ac.uk/depts/maths/tables/

[32] Max Lillack, Thorsten Berger, and Regina Hebig. 2016. Experiences from Reengineering and Modularizing a Legacy Software Generator with a Projectional Language Workbench. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC).*

[33] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and How to Develop Domain-specific Languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[34] Microsoft. 2022. *An early-stage PHP parser designed for IDE usage scenarios.* https://github.com/microsoft/tolerant-php-parser Accessed: 2022-04-10.

[35] Microsoft. 2022. *Language Server Protocol.* https://microsoft.github.io/language-server-protocol/ Accessed: 2022-04-05.

[36] Microsoft. 2022. *VSCode Language Server - Node.* https://github.com/microsoft/vscode-languageserver-node Accessed: 2022-04-10.

[37] M. Mészáros, M. Cserép, and A. Fekete. 2019. Delivering Comprehension Features into Source Code Editors through LSP. In *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* 1581–1586.

[38] Oracle. 2022. *Class TreePathScanner.* https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/util/TreePathScanner.html Accessed: 2022-04-10.

[39] Terence Parr. 2009. *Language Implementation Patterns: Create Your Own Domain-specific and General Programming Languages.* Pragmatic Bookshelf.

[40] Terence Parr. 2022. *ANTLR (ANother Tool for Language Recognition).* https://www.antlr.org/ Accessed: 2022-04-10.

[41] Karl Pearson. 1896. VII. Mathematical contributions to the theory of evolution. III. Regression, heredity, and panmixia. *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character* 187 (1896), 253–318.

[42] Sven Peldszus. 2022. *Security Compliance in Model-driven Development of Software Systems in Presence of Long-Term Evolution and Variants.* Springer. https://doi.org/10.1007/978-3-658-37665-9

[43] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. 2015. Incremental Co-Evolution of Java Programs based on Bidirectional Graph Transformation. In *Principles and Practices of Programming on The Java Platform (PPPJ)*, Ryan Stansifer and Andreas Krall (Eds.). ACM, 138–151. https://doi.org/10.1145/2807426.2807438

[44] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. 2016. Continuous Detection of Design Flaws in Evolving Object-oriented Programs Using Incremental Multi-pattern Matching. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE).* 578–589. https://doi.org/10.1145/2970276.2970338

[45] phpDocumentor. 2022. *ReflectionDocBlock.* https://github.com/phpDocumentor/ReflectionDocBlock Accessed: 2022-04-10.

[46] Angel Luis Scull Pupo, Jens Nicolay, Kyriakos Efthymiadis, Ann Nowé, Coen De Roover, and Elisa Gonzalez Boix. 2019. Guardiaml: Machine Learning-assisted Dynamic Information Flow Control. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 624–628.

[47] Python Code Quality Authority. 2022. *Pylint.* https://pypi.org/project/pylint/ Accessed: 2022-04-10.

[48] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2019. Geoscenario: An Open DSL for Autonomous Driving Scenario Representation. In *30th IEEE Intelligent Vehicles Symposium (IV).*

[49] RDocumentation. 2022. *cor: Correlation, Variance and Covariance (Matrices).* https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/cor Accessed: 2022-04-10.

[50] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. 2018. Towards a Language Server Protocol Infrastructure for Graphical Modeling. In *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18).* Association for Computing Machinery, New York, NY, USA, 370–380. https://doi.org/10.1145/3239372.3239383

[51] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. 2017. A Chrestomathy of DSL Implementations. In *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE).*

[52] SonarSource. 2022. *SonarLint Website.* https://www.sonarlint.org/ Accessed: 2022-04-10.
[53] Sourcegraph. 2022. *Langserver.org: A community-driven source of knowledge for Language Server Protocol implementations.* https://langserver.org/ Accessed: 2022-04-05.
[54] Diomidis Spinellis. 2001. Notable Design Patterns for Domain-specific Languages. *Journal of Systems and Software* 56, 1 (2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3
[55] Stardog Union. 2022. *StarDog IDE.* https://www.stardog.com/studio/ Accessed: 2022-04-10.
[56] Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. 2019. Language-independent Development Environment Support for Dynamic Runtimes. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019).* Association for Computing Machinery, New York, NY, USA, 80–90. https://doi.org/10.1145/3359619.3359746
[57] Daniel Strüber, Sven Peldszus, and Jan Jürjens. 2018. Taming Multi-Variability of Software Product Line Transformations. In *Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science, Vol. 10802)*, Alessandra Russo and Andy Schürr (Eds.). Springer, 337–355. https://doi.org/10.1007/978-3-319-89363-1_19
[58] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. 2010. *"Software Architecture - Foundations, Theory, and Practice".* Wiley.
[59] The Comprehensive R Archive Network. 2022. *arules: Mining Association Rules and Frequent Itemsets.* https://cran.r-project.org/web/packages/arules/index.html Accessed: 2022-04-10.
[60] The Rust Foundation. 2022. *Guide to Rustc Development: The HIR.* https://rustc-dev-guide.rust-lang.org/hir.html Accessed: 2022-04-10.
[61] M Völter, S Benz, C Dietrich, B Engelmann, M Helander, LCL Kats, E Visser, and GH Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-specific Languages.* M Volter / DSLBook.org.
[62] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-friendly Projectional Editors. In *7th ACM SIGPLAN International Conference on Software Language Engineering (SLE).*
[63] Markus Völter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *International Conference on Software Language Engineering (SLE).*
[64] Andrzej Wasowski and Thorsten Berger. 2022. *Domain-specific Languages: Effective Modeling, Automation, and Reuse.* http://dsl.design