Implementing a language Server for the Nickel Language

Prestudy

Student: Yannik Sander
Supervisor (Tweag): Yann Hamdaoui
Supervisor (KTH): Martin Monperrus
Examiner: Roberto Guanciale

May 29, 2022

Contents

1 Introduction

Integrated Development Environments (IDEs) and other more lightweight code editors are by far the most used tool of software developers. Yet, improvements of language intelligence, i.e. code completion, debugging as well as static code analysis refactoring and enrichment, have traditionally been subject to both the language and the editor used. Language support is thereby brought to IDEs by the means of platform dependent extensions that require repeated efforts for each platform and hence varied a lot in performance, feature-richness and availability. Recent years have seen different works [refs?] towards editor-independent code intelligence implementations and unified language-independent protocols. The to-date most successful approach is the Language Server Protocol (LSP). The protocol specifies how editors can communicate with language servers which are separate, editor independent implementations of language analyzers. It allows to express a wide variance of language intelligence. LSP servers allow editors to jump to definitions, find usages, decorate elements with additional information inline or when hovering elements, list symbols and much more. The LSP is discussed in more detail in sec. 2.1. These approaches reduce the effort required to bring language intelligence to editors. Instead of rewriting what is essentially the same language extension for every editor, any editor that implements an LSP client can connect to the same language specific server. Since LSP client implementations are independent of the servers, editor communities can focus on developing the best possible and uniform experience which all LSP servers can leverage. As a side effect, this also allows for developers to stay in their preferred developing environment instead of needing to resort to e.g., Vim or Emacs emulation or loosing access to other plugins.

Being independent of the editors, the developer of the language server is free to choose the optimal implementing language. In effect, it is possible for language developers to integrate essential parts of the existing language implementation for a language server. By now the LSP has become the most popular choice for cross-platform language tooling with implementations [langservers and microsoft] for all major and many smaller languages.

Speaking of smaller languages is significant, as both research communities and industry continuously develop and experiment with new languages for which tooling is unsurprisingly scarce. Additionally, previous research [ref], that shows the importance of language tools for the selection of a language, highlights the importance of tooling for new languages to be adopted by a wider community. While previously implementing language tools that integrate with the developer's environment was practically unfeasible for small projects due to the incompatibility between different extension systems, leveraging the LSP reduces the amount of work required considerably.

The Nickel**nickel?** language referenced in this work is a new Configuration Programming Language developed by Tweag. It is akin to projects like Cue, Dhall, or Nix in that it is an abstraction over pure data description languages such as JSON, YAML or XML. The Nickel project combines the capabilities of the former being a pure functional language based on lambda calculus with JSON data types, gradual typing, higher-order contracts and a record merging operation. As such, it is intended to write safe abstractions over configuration files as employed in Infrastructure as Code for instance.

1.1 Motivation

Since its release, the LSP has grown to be supported by a multitude of languages and editors**lsp-website?**, solving a long-standing problem with traditional IDEs.

Before the inception of language servers, implementing specialized features for every language of interest was the sole responsibility of the developers of code editors. Under the constraint of limited resources, editors had to position themselves on a spectrum between specializing on integrated support for a certain subset of languages and being generic over the language providing only limited support. As the former approach offers a greater business value, especially for proprietary products most professional IDEs gravitate towards excellent (and exclusive) support for single major languages, i.e., XCode and Visual Studio for the native languages for Apple and Microsoft Products respectively as well as JetBrains' IntelliJ platform and RedHat's Eclipse. Problematically, this results in less choice for developers and possible lock-in into products subjectively less favored but unique in their features for a certain language. The latter approach was taken by most text editors which in turn offered only limited support for any language.

Popularity statistics[1] from before the introduction of the LSP shows that except Vim and Sublime Text, both exceptional general text editors, the top 10 most popular IDEs were indeed specialized products. Regardless that some IDEs offer support for more languages through (third-party) extensions, developing any sort of language support to N platforms requires the implementation of N integrations. Missing standards, incompatible implementing languages and often proprietary APIs highlight this problem.

This is especially difficult for emerging languages, with possibly limited development resources to be put towards the development of language tooling. Consequently, efforts of language communities vary in scope, feature completeness and availability.

The Language Server Protocol aims to solve this issue by specifying an API that editors (clients) can use to communicate with language servers. Language servers are programs that implement a set of IDE features for one language and expose access to these features through the LSP. This allows developers to focus resources to a single project that is above all unrelated to editor-native APIs for analytics processing code representation and GUI integration. Now only a single implementation of a language server is required,

[1] https://web.archive.org/web/20160625140610/https://pypl.github.io/IDE.html

instead of an individual plugin for each editor. Editor maintainers can concentrate on offering the best possible LSP client support independent of the language.

1.1.1 Problem Definition

The problem this thesis will address is the current lack of documentation and evaluation of the applied methods for existing Language Servers.

While most of the implementations of LSP servers are freely available as Open Source Software [ref?], the methodology is often poorly documented, especially for smaller languages. There are some experience reports [ref: merlin, and others] and a detailed video series on the Rust Analyzer[ref or footnote] project, but implementations remain very opinionated and poorly guided through. The result is that new implementations keep repeating to develop existing solutions.

Moreover, most projects do not formally evaluate the Language Server on even basic requirements. Naïvely, that is, the server should be *performant* enough not to slow down the developer, it should offer *useful* information and capabilities and of course be *correct* as well as *complete*.

1.2 Research Questions

To guide future implementations of language servers for primarily small scale languages the research presented in this thesis aims to answer the following research questions at the example of the Nickel Project[2]:

RQ.1

a) How to develop a language server for a new language that

b) satisfies its users' needs while being performant enough not to slow them down?

RQ.2 How can we assess the implementation both quantitatively based on performance measures and qualitatively based on user satisfaction?

The goal of this research is to describe a reusable approach for representing programs that can be used to query data to answer requests on the Language Server Protocol efficiently. The research is conducted on an implementation of the open source language Nickel[ˆhttps://nickel-lang.org] which provides the *Diagnostics*, *Jump to ** and *Hover* features as well as limited *Auto-Completion* and *Symbol resolution*. Although implemented for and with integration of the Nickel runtime, the objective is to keep the internal format largely language independent. Similarly, the Rust based implementation should be

[2] https://nickel-lang.org

described abstractly enough to be implemented in other languages. To support the chosen approach, a user study will show whether the implementation is able to meet the expectations of its users and maintain its performance in real-world scenarios.

1.3 Non-Goals

The reference solution portrayed in this work is specific for the Nickel language. Greatest care is given to present the concepts as generically and transferable as possible. However, it is not a goal to explicitly cover a problem space larger than the Nickel language, which is a pure functional language based on lambda calculus with JSON data types, gradual typing, higher-order contracts and a record merging operation.

2 Background

This thesis illustrates an approach of implementing a language server for the Nickel language which communicates with its clients, i.e. editors, over the open Language Server Protocol (in the following abbreviated as *LSP*). The current chapter provides the background on the technological details of the project. As the work presented aims to be transferable to other languages using the same methods, this chapter will provide the means to distinguish the nickel specific implementation details.

The primary technology built upon in this thesis is the language server protocol. The first part of this chapter introduces the LSP, its rationale and improvements over classical approaches, technical capabilities and protocol details. The second part is dedicated to Nickel, elaborating on the context and use-cases of the language followed by an inspection of the technical features of Nickel.

2.1 Language Server Protocol

The Language Server Protocol is a JSON-RPC based communication specification comprising an LSP client (i.e. editors) and server (also called language server for simplicity). The LSP decouples the development of clients and servers, allowing developers to focus on either side. The LSP defines several capabilities – standardized functions which are remotely executed by the language server. LSP Clients are often implemented as editor extensions facilitating abstraction libraries helping with the interaction with the protocol and editor interface. Language Servers analyse source code sent by the client and may implement any number of capabilities relevant to the language. Since the LSP is both language and editor independent, the same server implementation can serve all LSP compliant clients eliminating the need to redundantly recreate the same code intelligence for every editor.

Language servers are today's standard of integrating support for programming languages into code editors. Initially developed by Microsoft for the use with their polyglot editor Visual Studio Code[1] before being released to the public in 2016 by Microsoft, RedHat and Codeenvy, the LSP decouples language analysis and provision of IDE-like features from the editor. Developed under open source license on GitHub[2], the protocol allows developers of editors and languages to work independently on the support for new

[1] https://code.visualstudio.com/
[2] https://github.com/microsoft/language-server-protocol/

languages. If supported by both server and client, the LSP now supports more than 24 language features[3] including code completion, code navigation facilities, contextual information such as types or documentation, formatting, and more.

2.1.1 JSON-RPC

the LSP uses JSON-RPC to communicate between a language server and a client. JSON-RPC (v2) [1] is a JSON based lightweight transport independent remote procedure call [2] protocol.

RPC is a paradigm that allows clients to virtually invoke a method at a connected process. The caller sends a well-defined message to a connected process which executes a procedure associated with the request, taking into account any transmitted arguments. Upon invoking a remote procedure, the client suspends the execution of its environment while it awaits an answer of the server, corresponding to a classical local procedure return.

JSON-RPC is a JSON based implementation of RPC. The protocol specifies the format and meaning of the messages exchanged using the protocol, in particular *Request* and *Notification* and *Result* messages. Since the protocol is based on JSON, is possible to be used through any text based communication channel which includes streams or network sockets. In fact, JSON-RPC only specifies a message protocol, leaving the choice of the transport method to the application.

All messages refer to a `method` and a set of `parameters`. Servers are expected to perform the same procedure associated with the requested `method` at any time. In return, clients have to follow the calling conventions for the requested method. Typically, messages are synchronous, i.e., the client awaits a result associated to the method and its parameters before it continues its execution of the calling environment. Hence, requests are marked with an `id` which is included in the result/error message necessarily sent by the server. If the client does not require a response, it can omit the `id` field. The message is then interpreted as a *notification*, which servers cannot respond to.

Part of the JSON-RPC specification is the ability for clients to batch requests and send a list of request or notification objects. In this case, the server should respond with a list of results matching each request, yet is free to process requests concurrently.

2.1.2 Commands and Notifications

The LSP builds on top of the JSON-RPC protocol described in the previous subsection. It defines four sets of commands:

[3] https://microsoft.github.io/language-server-protocol/specifications/specification-current/

The largest group are commands that are relevant in the scope of the currently opened document, e.g. autocompletion, refactoring, inline values and more. In total the LSP defines 33 [3] of such "Language Features." Editors will notify the server about file changes and client side interaction, i.e., opening, closing and renaming files using "Document Synchronization" methods. While most commands are defined in the document scope, i.e., a single actively edited file, the LSP allows clients to communicate changes to files in the opened project. This so called workspace comprises on or more root folders managed by the editor and all files contained in them. "Workspace Features" allow the server to intercept file creation, renaming or deletion to make changes to existing sources in other files. Use cases of these features include updating import paths, changing class names and other automations that are not necessary local to a single file. In addition, the LSP specifies so called "Window Features" which allow the server to control parts of the user interface of the connected editor. For instance, servers may instruct clients to show notifications and progress bars or open files.

2.1.3 Description of Key Methods

the authors of langserver.org [4] identified six "key methods" of the LSP. The methods represent a fundamental set of capabilities, specifically:

1. Code completion Suggest identifiers, methods or values at the cursor position.
2. Hover information Present additional information about an item under the cursor, i.e., types, contracts and documentation.
3. Jump to definition Find and jump to the definition of a local variable or identifier.
4. Find references List all usages of a defined variable.
5. Workspace/Document symbols List all variables in a workspace or document.
6. Diagnostics Analyze source code, i.e., parse and type check and notify the LSP Client if errors arise.

2.1.3.1 Code Completion

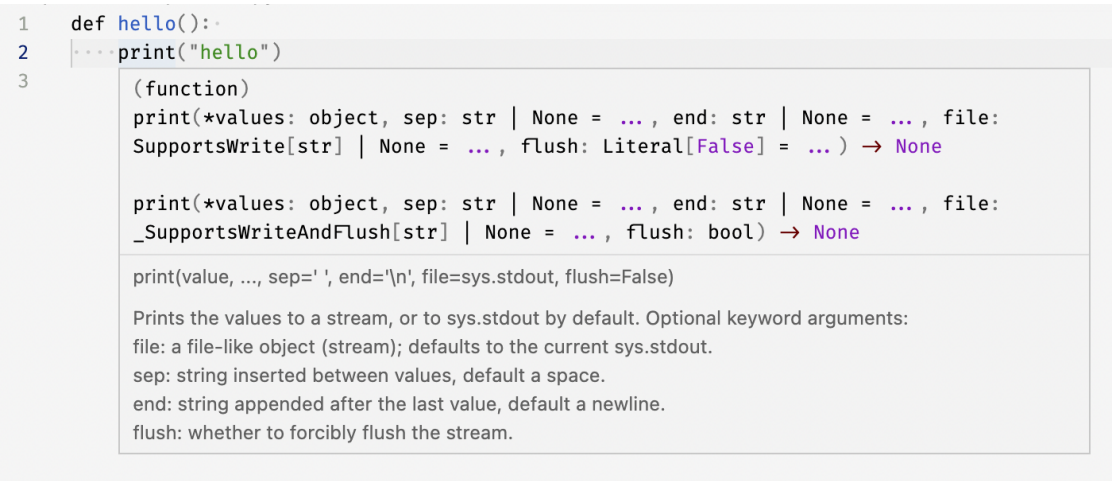
RPC Method: `textDocument/Completion`

This feature allows users to request a suggested identifier of variables or methods, concrete values or larger templates of generated code to be inserted at the position of the cursor. The completion can be invoked manually or upon entering language defined trigger characters, such as `.`, `::` or `->`. The completion request contains the current cursor position, allowing the language server to resolve contextual information based on an internal representation of the document. In the example (fig. **??**) the completion feature suggests related identifiers for the incomplete function call "`print`."

2.1.3.2 Hover

RPC Method: `textDocument/hover`

Hover requests are issued by editors when the user rests their mouse cursor on text in an opened document or issues a designated command in editors without mouse support. If the language server has indexed any information corresponding to the position, it can generate a report using plain text and code elements, which are then rendered by the editor. Language servers typically use this to communicate type-signatures or documentation. An example can be seen in fig. **??**.



2.1.3.3 Jump to Definition

RPC Method: `textDocument/definition`

The LSP allows users to navigate their code by the means of symbols by finding the definition of a requested symbol. Symbols can be for instance variable names or function calls. As seen in fig. **??**, editors can use the available information to enrich hover overlays with the hovered element's definition.

```
1  def hello(what):
2      (function) hello: (what: Any) → None
3
4      def hello(what):
5          print("hello", what)
6  hello("kitty")
7
```

2.1.3.4 Find References

RPC Method: `textDocument/references`

Finding references is the inverse operation to the previously discussed *Jump to Definition* (cf. 2.1.3.3). For a given symbol definition, for example variable, function, function argument or record field the LSP provides all usages of the symbol allowing users to inspect or jump to the referencing code position.

{#fig:lsp-capability-hover caption="Listing of all references to the method"hello". Python language server in Visual Studio Code"}

2.1.3.5 Workspace/Document symbols

RPC Method: `textDocument/workspaceSymbol` *or* `textDocument/documentSymbol`

The symbols capability is defined as both a "Language Feature" and "Workspace Feature" which mainly differ in the scope they represent. The `textDocument/documentSymbol` command lists symbols solely found in the currently opened file, while the `workspace/symbol` takes into account all files in the opened set of folders. The granularity of the listed items is determined by the server and possibly different for either scope. For instance, document symbols could be as detailed as listing any kind of method or property found in the document, while workspace symbols take visibility rules into account which might expose public entities only. Symbols are associated with a span of source code of the symbol itself and its context, for example a function name representing the function body. Moreover, the server can annotate the items with additional attributes such as symbol kinds, tags and even child-references (e.g. for the fields of a record or class).

2.1.3.6 Diagnostics

Diagnostics is the collective term for report statements about the analyzed language of varying severity. This can be parsing or compilation or type-checking n errors, as well as errors and warnings issued by a linting tool.

Unlike the preceding features discussed here, diagnostics are a passive feature, since most often the information stems from external tools being invoked after source code changes. File updates and diagnostics are therefore specified as notifications to avoid blocking the communication.

2.1.4 File Processing

Most language servers handling source code analysis in different ways. The complexity of the language can be a main influence for the choice of the approach. Distinctions appear in the way servers process *file indexes and changes* and how they respond to *requests*.

The LSP supports sending updates in form of diffs of atomic changes and complete transmission of changed files. The former requires incremental parsing and analysis, which are challenging to implement but make processing files much faster upon changes. An incremental approach makes use of an internal representation of the source code that allows efficient updates upon small changes to the source file.

Additionally, to facilitate the parsing, an incremental approach must be able to provide a parser with the right context to correctly parse a changed fragment of code. In practice, most language servers process file changes by re-indexing the entire file, discarding the previous internal state entirely. This is a more approachable method, as it poses less

requirements to the architects of the language server. Yet, it is far less performant. Unlike incremental processing (which updates only the affected portion of its internal structure), the smallest changes, including adding or removing lines effect the *reprocessing of the entire file*. While sufficient for small languages and codebases, non-incremental processing quickly becomes a performance bottleneck.

For code analysis LSP implementers have to decide between *lazy* or *greedy* approaches for processing files and answering requests. Dominantly greedy implementations resolve most available information during the indexing of the file. The server can then utilize this model to answer requests using mere lookups. This stands in contrast to lazy approaches where only minimal local information is resolved during the indexing. Requests invoke an ad-hoc resolution the results of which may be memoized for future requests. Lazy resolution is more prevalent in conjunction with incremental indexing, since it further reduces the work associated with file changes. This is essential in complex languages that would otherwise perform a great amount of redundant work.

2.2 Text based Configuration

Configuration languages such as XML[5], JSON[6], or YAML[7] are textual representations of structural data used to configure parameters of a system or application. The objective of such languages is to be machine readable, yet also intuitive enough for humans to write.

2.2.1 Common Configuration Languages

2.2.1.1 JSON

According to the JSON specification [6] the language was designed as a data-interchange format that is easy to read and write both for humans and machines. Since it is a subset of the JavaScript language, its use is particularly straight forward and wide spread in JavaScript based environments such as web applications. But due to its simplicity implementations are available and for all major languages, motivating its use for configuration files.

2.2.1.2 YAML

YAML is another popular language, mainly used for configuration files. According to its goals it should be human readable and independent of the programming language making use of it. It should be efficient to parse and easy to implement while being expressive and extensible.

YAML also features a very flexible syntax which allows for many alternative ways to declare semantically equivalent data. For example boolean expressions can be written as any of the following values:

```
y|Y|yes|Yes|YES|n|N|no|No|NO    true|True|TRUE|false|False|FALSE
on|On|ON|off|Off|OFF
```

Since YAML facilitates indentation as a way to express objects or lists and does not require object keys (and even strings) to be quoted, it is considered easier to write than JSON at the expense of parser complexity. Yet, YAML is compatible with JSON since as subset of its specification defines a JSON equivalent syntax that permits the use of `{..}` and `[..]` to describe objects and lists respectively.

2.2.2 Applications of Configuration Languages

Applications of configuration languages are ubiquitous especially in the vicinity of software development. While XML and JSON are often used by package managers [10], YAML is a popular choice for complex configurations such as CI/CD pipelines [11]–[13] or machine configurations in software defined networks such as Kubernetes[14] and docker compose [15].

Such formats are used due to some significant advantages compared to other binary formats such as databases. Most strikingly, the textual representation allows inspection of a configuration without the need of a separate tool but a text editor. Moreover textual configuration can be version controlled using VCS software like Git which allows changes to be tracked over time. Linux service configurations (files in `/etc`) and MacOS `*.plist` files which can be serialized as XML or a JSON-like format, especially exemplify that claim.

2.2.3 Configuration Programming Languages

Despite the above-mentioned formats being simple to parse and widely supported [6], their static nature rules out any dynamic content such as generated fields, functions and the possibility to factorize and reuse. Moreover, content validation has to be developed separately, which led to the design of complementary schema specification languages like json-schema [16] or XSD [17].

These qualities require an evaluated language. In fact, some applications make heavy use of config files written in the native programming language which gives them access to language features and existing analysis tools. Examples include JavaScript frameworks such as webpack [18] or Vue [19] and python package management using `setuptools`[20]. Yet, the versatility of general purpose languages poses new security risks if used in this context as configurations could now contain malicious code requiring additional

verification. Beyond this, not all languages serve as a configuration language, e.g. compiled languages.

However, for particularly complex applications, both advanced features and language independence are desirable. Alternatively to using high level general purpose languages, this demand is addressed by domain specific languages (DSL). Dhall [21], Cue [22] or jsonnet [23] are such domain specific languages, that offer varying support for string interpolation, (strict) typing, functions and validation. Most of these languages are used as a templating system which means a configuration file in a more portable format is generated using an evaluation of the more expressive configuration source. The added expressiveness manifests in the ability to evaluate expression and the availability of imports, variables and functions. These features allow to refactor and simplify repetitive configuration files.

2.2.4 Infrastructure as Code

The shift to an increasing application of IaaS[4] products started the desire for declarative machine configuration in a bid to simplify the setup and reproducibility of such systems. This configuration based setup of infrastructure is commonly summarized as infrastructure as code or IaC. As the name suggests, IaC puts cloud configuration closer to the domain of software development [24].

In principle, IaaS solutions offer great flexibility with regard to resource provision (computing, storage, load balancing, etc.), network setup and scaling of (virtual) servers. However, since the primary interaction with those systems is imperative and based on command line or web interfaces, maintaining entire applications' or company's environments manually comes with obvious drawbacks.

Changing and undoing changes to existing networks requires intricate knowledge about its topology which in turn has to be meticulously documented. Undocumented modification pose a significant risk for *config drift* which is particularly difficult to undo imperatively. Beyond that, interacting with a system through its imperative interfaces demands qualified skills of specialized engineers.

The concept of "Infrastructure as Code" (*IaC*) align with the principles of DevOps. IaC tools help to overcome the need for dedicated teams for *Dev*elopment and *Op*erations by allowing to declaratively specify the dependencies, topology and virtual resources. Optimally, different environments for testing, staging and production can be derived from a common base and changes to configurations are atomic. As an additional benefit, configuration code is subject to common software engineering tooling; It can be statically analyzed, refactored and version controlled to ensure reproducibility. A subset of IaC is focused on largely declarative configuration based on configuration files that are interpreted and "converted" into imperative platform dependent instructions.

[4] Infrastructure as a Service

As a notable instance, the Nix[[25];dolstraNixSafePolicyFree2004] ecosystem even goes as far as enabling declarative system and service configuration using NixOps[26].

To get an idea of how this would look like, lst. 2.1 shows the configuration for a deployment of the Git based wiki server Gollum[27] behind a nginx reverse proxy on the AWS network. Although this example targets AWS, Nix itself is platform-agnostic and NixOps supports different backends through various plugins. Configurations like this are abstractions over many manual steps and the Nix language employed in this example allows for even higher level turing-complete interaction with configurations.

Listing 2.1 Example NixOps deployment to AWS

```
{
  network.description = "Gollum server and reverse proxy";
  defaults =
    { config, pkgs, ... }:
    {
      # Configuration of a specific deployment implementation
      # here: AWS EC2

      deployment.targetEnv = "ec2";
      deployment.ec2.accessKeyId = "AKIA...";
      deployment.ec2.keyPair = "...";
      deployment.ec2.privateKey = "...";
      deployment.ec2.securityGroups = pkgs.lib.mkDefault [ "default" ];
      deployment.ec2.region = pkgs.lib.mkDefault "eu-west-1";
      deployment.ec2.instanceType = pkgs.lib.mkDefault "t2.large";
    };
  gollum =
    { config, pkgs, ... }:
    {
      # Nix based setup of the gollum server

      services.gollum = {
        enable = true;
        port = 40273;
      };
      networking.firewall.allowedTCPPorts = [ config.services.gollum.port ];
    };

  reverseproxy =
    { config, pkgs, nodes, ... }:
    let
      gollumPort = nodes.gollum.config.services.gollum.port;
    in
    {
      # Nix based setup of a nginx reverse proxy

      services.nginx = {
        enable = true;
        virtualHosts."wiki.example.net".locations."/" = {
          proxyPass = "http://gollum:${toString gollumPort}";
        };
      };
      networking.firewall.allowedTCPPorts = [ 80 ];

      # Instance can override default deployment options
      deployment.ec2.instanceType = "t1.medium";
    };
}
```

Similarly, tools like Terraform[28], or Chef[29] use their own DSLs and integrate with most major cloud providers. The popularity of these products[5], beyond all, highlights the importance of expressive configuration formats and their industry value.

Finally, descriptive data formats for cloud configurations allow mitigating security risks through static analysis. Yet, as recently as Fall 2021 dossiers of Palo Alto Networks' security department Unit 42 [30] show that a majority of public projects use insecure configurations. This suggests that techniques[31] to automatically check templates are not actively employed, and points out the importance of evaluated configuration languages which can implement passive approaches to security analysis.

2.2.5 Nickel

The Nickel[32] language is a configuration programming language as defined in sec. 2.2.3 with the aims of providing composable, verifiable and validatable configuration files. Nickel implements a pure functional language with JSON-like data types and turing-complete lambda calculus. The language draws inspiration from existing projects such as Cue [22], Dhall [21] and most importantly Nix [25]. However, Nickel sets itself apart from the existing projects by combining and advancing their strengths. The language addresses concerns drawn from the experiences with Nix which employs a sophisticated modules system to provide type-safe, composed (system) configuration files. Nickel implements gradual type annotations, with runtime checked contracts to ensure even complex configurations remain correct. Additionally, considering record merging on a language level facilitates modularization and composition of configurations.

2.2.5.1 Nickel AST

Nickel's syntax tree is a single sum type, i.e., an enumeration of node types. Each enumeration variant may refer to child nodes, representing a branch or hold terminal values in which case it is considered a leaf of the tree. Additionally, tree nodes hold information about their position in the underlying code.

2.2.5.1.1 Basic Elements The primitive values of the Nickel language are closely related to JSON. On the leaf level, Nickel defines `Boolean`, `Number`, `String` and `Null`. In addition to that the language implements native support for `Enum` values which are serialized as plain strings in less expressive formats such as JSON. Each of these are terminal leafs in the syntax tree.

Completing JSON compatibility, `List` and `Record` constructs are present as well. Records on a syntax level are Dictionaries, uniquely associating an identifier with a sub-node.

[5] https://trends.google.com/trends/explore?date=2012-01-01%202022-01-01&q=%2Fg%2F11g6bg27fp,CloudFormation

These data types constitute a static subset of Nickel which allows writing JSON compatible expressions as shown in lst. 2.2.

Listing 2.2 Example of a static Nickel expression

```
{
  list = [ 1, "string", null],
  "some key" = "value"
}
```

Beyond these basic elements, Nickel implements variables and functions as well as a special syntax for attaching metadata and recursive records.

2.2.5.1.2 Identifiers The inclusion of variables to the language, implies an understanding of identifiers. Such name bindings can be declared in multiple ways, e.g. `let` bindings, function arguments and records. The usage of a name is always parsed as a single `Var` node wrapping the identifier. Span information of identifiers is preserved by the parser and encoded in the `Ident` type.

Listing 2.3 Let bindings and functions in nickel

```
// simple bindings
let name = <expr> in <expr>
let func = fun arg => <expr> in <expr>

// or with patterns
let name @ { field, with_default = 2 } = <expr> in <expr>
let func = fun arg @ { field, with_default = 2 } =>
  <expr> in
  <expr>
```

2.2.5.1.3 Variable Reference Let bindings in their simplest form merely bind a name to a value expression and expose the name to the inner expression. Hence, the `Let` node contains the binding and links to both implementation and scope subtrees. The binding can be a simple name, a pattern or both by naming the pattern as shown in lst. 2.3.

Listing 2.4 Parsed representation of functions with multiple arguments

```
fun first second => first + second
// ...is parsed as
fun first =>
  fun second => first + second
```

Functions in Nickel are curried lambda expressions. A function with multiple arguments gets broken down into nested single argument functions as seen in lst. 2.4. Function argument name binding therefore looks the same as in `let` bindings.

2.2.5.1.4 Meta Information One key feature of Nickel is its gradual typing system [ref again?], which implies that values can be explicitly typed. Complementing type information, it is possible to annotate values with contracts and additional metadata such as contracts, documentation, default values and merge priority using a special syntax as displayed in lst. 2.5.

Listing 2.5 Example of a static Nickel expression

```
let Contract = {
  foo | Num
      | doc "I am foo",
  hello | Str
        | default = "world"
}
| doc "Just an example Contract"
in
let value | #Contract = { foo = 9, }
in value == { foo = 9, hello = "world", }

> true
```

Internally, the addition of annotations wraps the annotated term in a `MetaValue`, an additional tree node which describes its subtree. The expression shown in lst. 2.6 translates to the AST in fig. 2.1.

Listing 2.6 Example of a typed expression

```
let x: Num = 5 in x
```

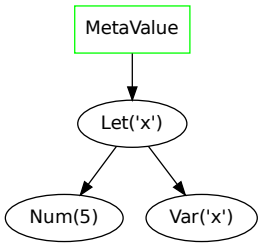


Figure 2.1: AST of typed expression

2.2.5.1.5 Nested Record Access Nickel supports both static and dynamic access to record fields. If the field name is statically known, the access is said to be *static* accordingly. Conversely, if the name requires evaluating a string from an expression the access is called *dynamic*. An example is given in lst. 2.7

Listing 2.7 Examples for static and dynamic record access

```
let r = { foo = 1, "bar space" = 2} in
r.foo // static
r."bar space" // static
let field = "fo" ++ "o" in r."#{field}" // dynamic
```

The destruction of record fields is represented using a special set of AST nodes depending on whether the access is static or dynamic. Static analysis does not evaluate dynamic fields and thus prevents the analysis of any deeper element starting with dynamic access. Static access however can be used to resolve any intermediate reference.

Notably, Nickel represents static access chains in inverse order as unary operations which in turn puts the terminal `Var` node as a leaf in the tree. Figure 2.2 shows the representation of the static access performed in lst. 2.8 with the rest of the tree omitted.

Listing 2.8 Nickel static access

```
let x = {
  y = {
    z = 1,
  }
} in x.y.z
```

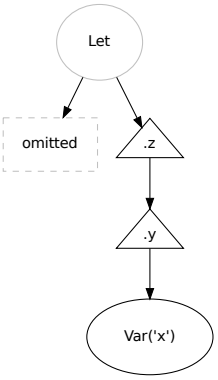


Figure 2.2: AST of typed expression

2.2.5.1.6 Record Shorthand Nickel supports a shorthand syntax to efficiently define nested records similarly to how nested record fields are accessed. As a comparison the example in lst. 2.9 uses the shorthand syntax which resolves to the semantically equivalent record defined in lst. 2.10

Listing 2.9 Nickel record defined using shorthand

```
{
  deeply.nested.record.field = true,
}
```

Listing 2.10 Nickel record defined explicitly

```
{
  deeply = {
    nested = {
      record = {
        field = true,
      }
    }
  }
}
```

Yet, on a syntax level Nickel generates a different representation.

2.2.5.2 Record Merging

Nickel considers record merging as a fundamental operation that combines two records (i.e. JSON objects). Merging is a commutative operation between two records which takes the fields of both records and returns a new record that contains the fields of both operands (cf. lst. 2.11)

Listing 2.11 Merging of two records without shared fields

```
{ enable = true } & { port = 40273 }
>>
{
  enable = true,
  port = 40273
}
```

If both operands contain a nested record referred to under the same name, the merging operation will be applied to these records recursively (cf. lst. 2.12).

Listing 2.12 Merging of two records without shared nested records

```
let enableGollum = {
  service = {
    gollum = {
      enable = true
    }
  }
} in

let setGollumPort = {
  service = {
    gollum = {
      port = 40273
    }
  }
} in

enableGollum & setGollumPort

>>
{
  service = {
    gollum = {
      enable = true,
      port = 40273
    }
  }
}
```

However, if both operands contain a field with the same name that is not a mergeable record, the operation fails since both fields have the same priority making it impossible for Nickel to chose one over the other (cf. lst. 2.13) Specifying one of the fields as a `default` value allows a single override (cf. lst. 2.14). In future versions of Nickel ([33]) it will be possible to specify priorities in even greater detail and provide custom merge functions.

Listing 2.13 Failing merge of two records with common field

```
{ port = 40273 } & { port = 8080 }

>>
error: non mergeable terms
  |
1 | { port = 40273 } & { port = 8080 }
  |          ^^^^^              ^^^^ with this expression
  |          |
  |          cannot merge this expression
```

Listing 2.14 Succeeding merge of two records with default value for common field

```
{ port | default = 40273 } & { port = 8080 }

>>
{ port = 8080 }
```

2.2.5.3 Gradual typing

The typing approach followed by Nickel was introduce by Siek and Taha [34] as a combination of static and dynamic typing. The choice between both type systems is traditionally debated since either approach imposes specific drawbacks. Static typing lacks the flexibility given by fully dynamic systems yet allow to ensure greater correctness by enforcing value domains. While dynamic typing is often used for prototyping, once an application or schema stabilizes, the ability to validate data schemas is usually preferred, often requiring the switch to a different statically typed language. Gradual typing allows introducing statically checked types to a program while allowing other parts of the language to remain untyped and thus interpreted dynamically.

2.2.5.4 Contracts

In addition to a static type-system Nickel integrates a contract system akin what is described in [35]. First introduced by Findler and Felleisen, contracts allow the creation of runtime-checked subtypes. Unlike types, contracts check an annotated value using arbitrary functions that either pass or *blame* the input. Contracts act like assertions that are automatically checked when a value is used or passed to annotated functions.

For instance, a contract could be used to define TCP port numbers, like shown in lst. 2.15.

Listing 2.15 Sample Contract ensuring that a value is a valid TCP port number

```
let Port | doc "A contract for a port number" =
  contracts.from_predicate (
    fun value =>
      builtins.is_num value &&
      value % 1 == 0 &&
      value >= 0 &&
      value <= 65535
  )
in 8080 | #Port
```

Going along gradual typing, contracts pose a convenient alternative to the `newtype` pattern. Instead of requiring values to be wrapped or converted into custom types, contracts are self-contained. As a further advantage, multiple contracts can be applied to the same value as well as integrated into other higher level contracts. An example can be observed in lst. 2.16

Listing 2.16 More advaced use of contracts restricting values to an even smaller domain

```
let Port | doc "A contract for a port number" =
  contracts.from_predicate (
    fun value =>
      builtins.is_num value &&
      value % 1 == 0 &&
      value >= 0 &&
      value <= 65535
  )
in
let UnprivilegedPort = contracts.from_predicate (
  fun value =>
    (value | #Port) >= 1024
  )
in
let Even = fun label value =>
  if value % 2 == 0 then value
  else
    let msg = "not an even value" in
    contracts.blame_with msg label
in

8001 | #UnprivilegedPort
     | #Even
```

Notice how contracts also enable detailed error messages (see lst. 2.17) using custom blame messages. Nickel is able to point to the exact value violating a contract as well as the contract in question.

Listing 2.17 Example error message for failed contract

```
error: Blame error: contract broken by a value [not an even value].
    - :1:1
   |
 1 | #Even
   | ----- expected type
   |
    - repl-input-34:22:1
   |
22 | - 8001 | #UnprivilegedPort
   |   ---- evaluated to this expression
23 | |      | #Even
   | -------------^ applied to this expression

note:
    - repl-input-34:23:8
   |
23 |      | #Even
   |        ^^^^^ bound here
```

3 Related work

The Nickel Language Server follows a history of previous research and development in the domain of modern language tooling and editor integration. Most importantly, it is part of a growing set of LSP integrations. As such, it is important to get a picture of the field of current LSP projects and the approaches leading up to the LSP. This chapter will first introduce and show the shortcomings of classical IDE based language tooling as well as early advances towards editor independent implementations. The following section displays particular instances of different approaches towards the implementation of language servers. Projects that extend and supplement or take inspiration from the LSP, are portrayed in the final section.

3.1 IDE Support

3.1.1 Native and Plugin Systems

Before the invention of the Language Server Protocol, language intelligence used to be provided by an IDE. Yet, the range of officially supported languages remained relatively small[1]. While integration for popular languages was common, top-tier support for less popular ones was all but guaranteed and relied mainly on community efforts. In fact Eclipse[37], IntelliJ[38], and Visual Studio[39], to this day the most popular IDE choices, focus on a narrow subset of languages, historically Java and .NET. Additional languages can be integrated by custom (third-party) plugins or derivatives of the base platform. Due to the technical implications, plugins are generally not compatible between different platforms. Many less popular languages therefore saw redundant implementations of what is essentially the same. For Haskell separate efforts produced an eclipse based IDE[40], as well as independent IntelliJ plugins[42]. Importantly, the implementers of the former reported troubles with the language barrier between Haskell and the Eclipse base written in Java.

The Haskell language is an exceptional example since there is also a native Haskell IDE[43] albeit that it is available only to the MacOS operating system. This showcases the difficulties of language tooling and its provision, since all of these projects are platform dependent and differ in functionality. Moreover, effectively the same tool is developed multiple times wasting resources.

[1] https://www.jetbrains.com/products/

In general, developing language integrations, both as the vendor of an IDE or a third-party plugin developer requires extensive resources. Table 3.1 gives an idea of the efforts required. Since the IntelliJ platform is based on the JVM, its plugin system requires the use of JVM languages[44], making it hard to reuse the code of e.g. a reference compiler or interpreter. The Rust and Haskell integrations for instance contain at best only a fraction of code in their respective language.

Table 3.1: Comparison of the size for different IntelliJ platform plugins

Plugin	lines of code
intellij-haskell	17249 (Java) + 13476 (Scala) **+ 0 (Haskell)**
intellij-rust	229131 (Kotlin) + 3958 (Rust)
intellij-scala	39382 (Java) + 478904 (Scala)
intellij-kotlin	182372 (Java) + 563394 (Kotlin)
intellij-community/python	47720 (C) + 248177 (Java) + 37101 (Kotlin) + 277125 (Python)

Naturally, development efforts at this size tend to gravitate around the most promising solution, stifling the progress on competing platforms. Editor-specific approaches also tend to lock-in programmers into a specific platform for its language support regardless of their personal preference.

3.1.2 Server Client Abstractions

3.1.2.1 Monto

The authors of the Monto project[46] call this the "IDE Portability Problem." They compare the situation with the task of compiling different high level languages to a set of CPU architectures. In Compilers, the answer to that problem was the use of an intermediate representation (IR). A major compiler toolchain making use of this is the LLVM[47]. Compiler frontends for different languages – e.g. Clang[48], Rustc[49], NVCC[50],... – compile input languages into LLVM IR, a low level language with additional capabilities to provide optimization hints but independent of a particular architecture. LLVM performs optimization of the IR and passes it to a compiler backend which in turn generates bytecode for specific architectures e.g. `x86_64`, `MIPS`, `aarch64`, `wasm`, etc. Notably through this mechanism, languages gain support for a range of architectures and profit from existing optimizations developed for the IR.

With Monto, Kreidel et al propose a similar idea for IDE portability. The paper describes the *Monto IR* and how they use a *Message Broker* to receive events from the Editor and dispatch them to *Monto Services*.

The Monto IR is a language-agnostic and editor-independent tree-like model serialized as JSON. Additionally, the IR maintains low level syntax highlighting information (font, color, style, etc.) but leaves the highlighting to the language specific service.

The processing and modification of the source code and IR is performed by *Monto Services*. Services implement specific actions, e.g. parsing, outlining or highlighting. A central broker connects the services with each other and the editor.

Since Monto performs all work on the IR, independent of the editor, and serializes the IR as JSON messages, the language used to implement *Monto Services* can be chosen freely giving even more flexibility.

The Editor extension's responsibility is to act as a source and sink for data. It sends Monto compliant messages to the broker and receives processing results such as (error) reports. The communication is based on the ZeroMQ[51] technology which was chosen because it is lightweight and available in manly languages[46] allowing to make use of existing language tools.

3.1.2.2 Merlin

The Merlin tool[53] is in many ways a more specific version of the idea presented in Monto. Merlin is a language server for the Ocaml language, yet predates the Language Server Protocol.

The authors of Merlin postulate that implementing "tooling support traditionally provided by IDEs" for "niche languages" demands to "share the language-awareness logic" between implementations. As an answer to that, they describe the architecture of Merlin in [53].

Similarly to Monto, Merlin separates editor extensions from language analysis. However, Merlin uses a command line interface instead of message passing for interaction. Editor extensions expose the server functions to the user by integrating with the editor.

Thanks to this architecture, the Merlin developers have been able to focus their efforts on a single project providing intelligence for OCaml source code. The result of this work is a platform independent, performant and fault-tolerant provider of language intelligence. Low level changes to the compiler core have been made to provide incremental parsing, type-checking and analysis. Apart from more efficient handling of source changes, this allows users to query information even about incomplete or incorrect programs.

Since both Merlin and the OCaml compiler are written in OCaml, Merlin is able to reuse large parts of the reference OCaml implementation. This allows Merlin to avoid reimplementing every single feature of the language. Still, incremental parsing and typechecking is not a priority to the compiler which prompted the developers of Merlin to vendor modified versions of the core OCaml components.

While Merlin serves as a single implementation used by all clients, unlike Monto it does not specify a language independent format, or service architecture. In fact, Merlin

explicitly specializes in a single language and provides a complete implementation where Monto merely defines the language agnostic interface to implement a server on.

3.2 Language Servers

The LSP project was announced[55] in 2016 to establish a common protocol over which language tooling could communicate with editors. The LSP helps the language intelligence tooling to fully concentrate on source analysis instead of integration with specific editors by building custom GUI elements and being restricted to editors extension interface.

At the time of writing the LSP is available in version ‘3.16[3]. Microsoft's official website lists 172 implementations of the LSP[56] for an equally impressive number of languages.

An in-depth survey of these is outside the scope of this work. Yet, a few implementations stand out due to their sophisticate architecture, features, popularity or closeness to the presented work.

Since the number of implementations of the LSP is continuously growing, this thesis will present a selected set of notable projects. The presented projects exemplify different approaches with respect to reusing and interacting with the existing language implementation of the targeted language. In particular, the following five approaches are discussed:

1. Three complete implementations that tightly integrate with the implementation level tooling of the respective language: *rust-analyzer[57],* ocaml-lsp/merlin[53] and the *Haskell Language Server[59]
2. A project that indirectly interacts with the language implementation through an interactive programming shell (REPL). *Frege LSP[61]*
3. A Language Server that is completely independent of the target language's runtime. Highlighting how basic LSP support can be implemented even for small languages in terms of userbase and complexity. *rnix-lsp[63]
4. Two projects that facilitate the LSP as an interface to an existing tool via HTTP or command line. *CPAchecker[65] and* CodeCompass[66]
5. An approach to generate language servers from domain specific language specifications[67].

3.2.1 Integrating with the Compiler/Runtime

Today LSP-based solutions serve as the go-to method to implement language analysis tools. Emerging languages in particular take advantage from the flexibility and reach of the LSP. Especially the freedom of choice for the implementing language, is facilitated by multiple languages by integrating modules of the original compiler or runtime into the language server.

3.2.1.1 HLS

For instance the Haskell language server facilitates a plugin system that allows it to integrate with existing tooling projects[2]. Plugins provide capabilities for linting, formatting, documentation and other code actions across multiple compiler versions. This architecture allows writing an LSP in a modular fashion in the targeted language at the expense of requiring HSL to use the same compiler version in use by the IDE and its plugins. This is to ensure API compatibility between plugins and the compiler.

3.2.1.2 Ocaml LSP

Similarly, the Ocaml language service builds on top of existing infrastructure by relying on the Merlin project introduced insec. **??**. Here, the advantages of employing existing language components have been explored even before the LSP.

3.2.1.3 Rust-Analyzer

The rust-analyzer[57] takes an intermediate approach. It does not reuse or modify the existing compiler, but instead implements analysis functionality based on low level components. This way the developers of rust-analyzer have greater freedom to adapt for more advanced features. For instance rust-analyzer implements an analysis optimized parser with support for incremental processing. Due to the complexity of the language, LSP requests are processed lazily, with support for caching to ensure performance. While many parts of the language have been reimplemented with a language-server-context in mind, the analyzer did not however implement detailed linting or the rust-specific borrow checker. For these kinds of analysis, rust-analyzer falls back to calls to the rust build system.

3.2.1.4 Frege LSP

While the previous projects integrated into the compiler pipeline and processed the results separately, other approaches explored the possiblity to shift the entire analysis to existing modules. A good example for this method is given by the Frege language[3].

Frege as introduced in[60] is a JVM based functional language with a Haskell-like syntax. It features lazy evaluation, user-definable operators, type classes and integration with Java.
While previously providing an eclipse plugin[4], the tooling efforts have since been put towards an LSP implementation. The initial development of this language server has

[2]https://hackage.haskell.org/packages/browse?terms=hls
[3]https://github.com/Frege/frege
[4]https://github.com/Frege/eclipse-plugin

been reported on in[61]. The author shows though multiple increments how they utilized the JVM to implement a language server in Java for the (JVM based) Frege language. In the final proof-of-concept, the authors build a minimal language server through the use of Frege's existing REPL and interpreter modules. The file loaded into the REPL environment providing basic syntax and type error reporting. The Frege LSP then translates LSP requests into expressions, evaluates them in the REPL environment and wraps the result in a formal LSP response. Being written in Java, allows the server to make use of other community efforts such as the LSP4J project which provide abstractions over the interaction with LSP clients. Through the use of abstraction like the Frege REPL, servers can focus on the implementation of capabilities only, albeit with the limits set by the interactive environment.

3.2.1.5 Runtime-independent LSP implementations

While many projects do so, language servers do not need to reuse any existing infrastructure of a targeted language at all. Often, language implementations do not expose the required language interfaces (parsing, AST, Types, etc..), or pose various other impediments such as a closed source, licensing, or the absence of LSP abstractions available for the host language.

An instance of this type is the rnix-lsp[63] language server for the Nix[62] programming language. Despite the Nix language being written in C++[25], its language server builds on a custom parser called "rnix[68] in Rust. However, since rnix does not implement an interpreter for nix expressions the rnix based language server is limited to syntactic analysis and changes.

3.2.1.6 Language Server as an Interface to CLI tools

While language servers are commonly used to provide code based analytics and actions such as refactoring, it also proved suitable as a general interface for existing external tools. These programs may provide common LSP features or be used to extend past the LSP.

3.2.1.7 CPAchecker

The work presented by Leimeister in[65] exemplifies how LSP functionality can be provided by external tools. The server can be used to automatically perform software verification in the background using CPAchecker[64]. CPAchecker is a platform for automatic and extensible software verification. The program is written in Java and provides a command line interface to be run locally. Additionally, it is possible to execute resource intensive verification through an HTTP-API on more powerful machines or clusters[70]. The LSP server supports both modes of operation. While it can interface directly with the Java

modules provided by the CPAchecker library, it is also able to utilize an HTTP-API provided by a server instance of the verifier.

3.2.1.8 CodeCompass

Similar to the work by Leimeister (c.fsec. 3.2.1.7), in[71] Mészáros et al. present a proof of concept leveraging the LSP to integrate (stand-alone) code comprehension tools with the LSP compliant VSCode editor. Code comprehension tools support the work with complex code bases by "providing various textual information, visualization views and source code metrics on multiple abstraction levels." Pushing the boundaries of LSP use-cases, code comprehension tools do not only analyze specific source code, but also take into account contextual information. One of such tools is CodeCompass[66]. The works of Mészáros yielded a language server that allowed to access the analysis features of CodeCompass in VSCode. In their paper they specifically describe the generation of source code diagrams. Commands issued by the client are processed by a CodeCompass plugin which acts as an LSP server and interacts with CodeCompass through internal APIs.

3.2.2 Language Servers generation for Domain Specific Languages

Bünder and Kuchen[67] highlight the importance of the LSP in the area of Domain Specific Languages (DSL). Compared to general purpose languages, DSLs often targets both technical and non-technical users. While DSL creation workbenches like Xtext[72], Spoofax[73] or MPS[74] allow for the implementation and provision of Eclipse or IntelliJ based DSLs, tooling for these languages is usually tied to the underlying platform. Requiring a specific development platform does not satisfy every user of the language. Developers have their editor of choice, that they don't easily give up on. Non-technical users could easily be overwhelmed by a complex software like Eclipse. For those non-technical users, a light editor would be more adapted, or even one that is directly integrated into their business application. The authors of[67] present how Xtext can generate an LSP server for a custom DSL, providing multi-editor support. The authors especially mention the Monaco Editor[75], a reusable HTML component for code editing using web technologies. It is used in products like VSCode[5], Theia[76] and other web accessible code editors. The Monaco Editor supports the LSP as a client (that is, on the editor side). Such LSP-capable web editors make integrating DSLs directly into web applications easier than ever before.

[5] https://github.com/Frege/eclipse-plugin

3.3 Alternative approaches

3.3.1 LSP Extensions

The LSP defines a large range of commands and capabilities which is continuously being extended by the maintainers of the protocol. Yet, occasionally server developers find themselves in need of functionality not yet present in the protocol. For example the LSP does not provide commands to exchange binary data such as files. Insec. **??** the CodeCompass Language Server was introduced. A stern feature of this server is the ability to generate and show diagrams in SVG format. However, the LSP does not define the concept of *requesting diagrams*. In particular Mészáros et al. describe different shortcomings of the LSP :

1. "LSP doesn't have a feature to place a context menu at an arbitrary spot in the document"

   Context menu entries are implemented by clients based on the agreed upon capabilities of the server. Undefined capabilities cannot be added to the context menu.

   In the case of CodeCompass the developers made up for this by using the code completion feature as an alternative dynamic context menu.

2. "LSP does not support displaying pictures (diagrams)."

   CodeCompass generates diagrams for selected code. Yet, there is no image transfer included with the LSP. Since the LSP is based on JSON-RPC messages, the authors' solution was to define a new command, specifically designed to tackle this non-standard use case.

Missing features of the protocol such as the ones pointed out by Mészáros et al. appear frequently, especially in complex language servers or ones that implement more than basic code processing.

The rust-analyzer defines almost thirty non-standard commands[6], to enable different language specific actions.

Taking the idea of the CodeCompass project further, Rodriguez-Echeverria et al. propose a generic extension of the LSP for graphical modeling[77]. Their approach is based on a generic intermediate representation of graphs which can be generated by language servers and turned into a graphical representation by the client implementation.

Similarly, in[79] the authors describe a method to develop language agnostic LSP extensions. In their work they defined a language server protocol for specification languages (SLSP) which builds on top of the existing LSP. The SLSP defines several extensions that each group the functionality of specific domains. However, unlike other LSP extensions

[6] https://rust-analyzer.github.io/manual.html#features

that are added to facilitate functions of a specific server, SLSP is language agnostic. In effect, the protocol extensions presented by Rask et. al. are reusable across different specification languages, allowing clients to implements a single frontend. Given their successes with the presented work, the authors encourage to build abstract, sharable extensions over language specific ones if possible.

3.3.2 Language Server Index Format

Nowadays, code hosting platforms are an integral part of the developer toolset (GitHub, Sourcegraph, GitLab, Sourceforge, etc.). Those platforms commonly display code simply as text, highlighted at best. LSP-like features would make for a great improvement for code navigation and code reading online. Yet, building these features on language servers would incur redundant and wasteful as a server needed to be started each time a visitor loads a chunk of code. Since the hosted code is most often static and precisely versioned, code analysis could be performed ahead of time, for all files of each version. The LSIF (Language Server index Format) specifies a schema for the output of such ahead of time code analysis. Clients can then provide efficient code intelligence using the pre-computed and standardized index.

The LSIF format encodes a graphical structure which mimics LSP types. Vertices represent higher level concepts such as `document`s, `range`s, `resultSet`s and actual results. The relations between vertices are expressed through the edges.

For instance, hover information as introduced insec. 2.1.3.2 for the interface declaration inlst. 3.1 can be represented using the LSIF. Figure 3.1 visualizes the result (cflst. 3.2). Using this graph an LSIF tool is able to resolve statically determined hover information by performing the following steps.

1. Search for `textDocument/hover` edges.
2. Select the edge that originates at a `range` vertex corresponding to the requested position.
3. Return the target vertex.

Listing 3.1 Exemplary code snippet to showing LSIF formatting

```
export interface ResultSet {
}
```

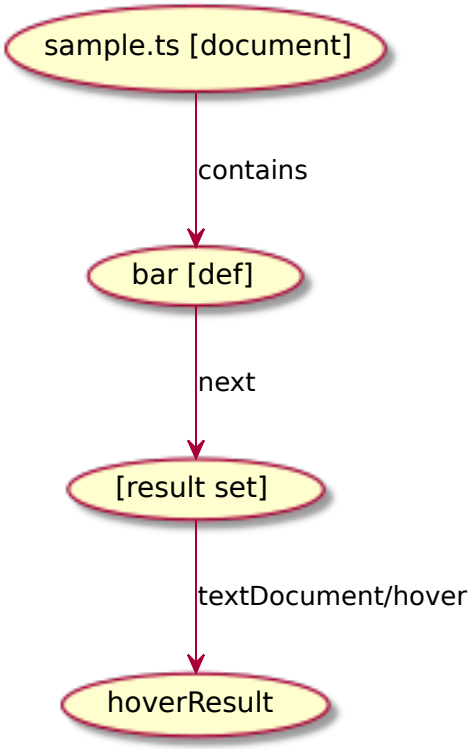


Figure 3.1: LSIF encoded graph for the exemplary code

Listing 3.2 LSIF formated analysis result

```
{ id: 1, type: "vertex", label: "document", uri: "file:///...", languageId: "typescript"
{ id: 2, type: "vertex", label: "resultSet" }
{
   id: 3,
  type: "vertex",
  label: "range",
  start: { line: 0, character: 9},
  end: { line: 0, character: 12 }
}
{ id: 4, type: "edge", label: "contains", outV: 1, inVs: [3] }
{ id: 5, type: "edge", label: "next", outV: 3, inV: 2 }
{
  id: 6,
  type: "vertex",
  label: "hoverResult",
  result: {
    "contents":[
      {"language":"typescript","value":"function bar(): void"},
      ""
    ]
  }
}
{ id: 7, type: "edge", label: "textDocument/hover", outV: 2, inV: 6 }
```

An LSIF report is a mere list of `edge` and `vertex` nodes, which allows it to easily extend and connect more subgraphs, corresponding to more elements and analytics. As a consequence, a subset of LSP capabilities can be provided statically based on the preprocessed LSIF model.

3.3.3 *SP, Abstracting software development processes

Since its introduction the Language Server Protocol has become a standard format to provide language tooling for editing source code. Meanwhile, as hinted insec. 3.3.1, the LSP is not able to fully satisfy every use-case sparking the development of various LSP extensions. Following the success of language servers, similar advances have been made in other parts of the software development process.

For instance, many Java build tools expose software build abstractions through the Build Server Protocol[80], allowing IDEs to integrate more languages more easily by leveraging the same principle as the LSP. The BSP provides abstractions over dependencies, build

targets, compilation and running of projects. While the LSP provides `run` or `test` integration for selected languages through Code Lenses, this is not part of the intended responsibilities of the protocol. In contrast, those tasks are explicitly targeted by the BSP.

Next to *writing* software (LSP) and *building/running/testing* software (e.g. BSP), *debugging* presents a third principal task of software development. Similar to the other tasks, most actions and user interfaces related to debugging are common among different languages (stepping in/out of functions, pausing/continuing exection, breakpoints, etc.). Hence, the Debug Adapter Protocol, as maintained by Microsoft and implemented in the VSCode Editor, aims to separate the language specific implementation of debuggers from the UI integration. Following the idea of the LSP, the DAP specifies a communication format between debuggers and editors. Since debuggers are fairly complicated software, the integration of editor communication should not prompt new developments of debuggers. Instead, the DAP assumes a possible intermediate debugger adapter do perform and interface with existing debuggers such as `LLDB`, `GDB`, `node-debug` and others[81].

Following the named protocols, Jeanjean et al. envision a future[82] where all kinds of software tools are developed as protocol based services independent of and shared by different IDEs and Editors. Taking this idea further, they call for a Protocol Specification that allows to describe language protocols on a higher level. Such a protocol, they claim, could enable editor maintainers to implement protocol clients more easily by utilizing automated generation from Language Service Protocol Specifications. Additionally, it could allow different Language Services to interact with and depend on other services.

References

[1] "JSON-RPC 2.0 Specification," JSON-RPC Working Group, Specification, Jan. 2013 [Online]. Available: https://www.jsonrpc.org/specification. [Accessed: Dec. 15, 2021]

[2] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, p. 21, 1984.

[3] Microsoft, "Language Server Protocol Specification - 3.16." Dec. 2020 [Online]. Available: https://microsoft.github.io/language-server-protocol/specification.html. [Accessed: Jul. 28, 2021]

[4] Sourcegraph, "Langserver.org." [Online]. Available: https://langserver.org/. [Accessed: Apr. 27, 2022]

[5] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," Specification, Nov. 2008 [Online]. Available: https://www.w3.org/TR/2008/REC-xml-20081126/. [Accessed: May 24, 2022]

[6] P. Charollais, "ECMA-404, 2nd edition, December 2017," p. 16, 2017.

[7] "YAML Ain't Markup Language (YAML™) revision 1.2.2" [Online]. Available: https://yaml.org/spec/1.2.2/. [Accessed: May 24, 2022]

[8] C. R. Pereira, "Managing modules with NPM," in *Building APIs with Node.js*, Berkeley, CA: Apress, 2016, pp. 9–13 [Online]. Available: https://doi.org/10.1007/978-1-4842-2442-7_3

[9] B. Varanasi and S. Belida, "Maven project basics," in *Introducing maven*, Berkeley, CA: Apress, 2014, pp. 23–36 [Online]. Available: https://doi.org/10.1007/978-1-4842-0841-0_4

[10] N. Adermann and J. Boggiano, "Composer." [Online]. Available: https://getcomposer.org/. [Accessed: May 24, 2022]

[11] R. Vemula, "Configuring a continuous build with travis CI," in *Real-time web application development : With ASP.NET core, SignalR, docker, and azure*, Berkeley, CA: Apress, 2017, pp. 489–508 [Online]. Available: https://doi.org/10.1007/978-1-4842-3270-5_13

[12] P. Heller, *Automating Workflows with GitHub Actions: Automate software development workflows and seamlessly deploy your applications using GitHub Actions*. Packt Publishing, 2021.

[13] M. S. Arefeen and M. Schiller, "Continuous integration using gitlab," *Undergraduate Research in Natural and Clinical Science and Technology Journal*, vol. 3, pp. 1–6, 2019.

[14] B. Burns, J. Beda, and K. Hightower, "Kubernetes: Up and Running," p. 277.

[15] K. Jangla, "Docker compose," in *Accelerating development velocity using docker: Docker across microservices*, Berkeley, CA: Apress, 2018, pp. 77–98 [Online]. Available: https://doi.org/10.1007/978-1-4842-3936-0_6

[16] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," in *Proceedings of the 25th International Conference on World Wide Web*, Apr. 2016, pp. 263–273, doi: 10.1145/2872427.2883029 [Online]. Available: https://doi.org/10.1145/2872427.2883029. [Accessed: May 24, 2022]

[17] W3C, "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures." [Online]. Available: https://www.w3.org/TR/xmlschema11-1/. [Accessed: Jan. 02, 2022]

[18] "Webpack." JS Foundation [Online]. Available: https://webpack.js.org/. [Accessed: May 24, 2022]

[19] C. Macrae, *Vue.js: Up and Running: Building Accessible and Performant Web Apps*, 1st edition. Sebastopol, California: O'Reilly Media, 2018.

[20] A. Martelli, A. Ravenscroft, and S. Holden, *Distributing Extensions and Programs*. [Online]. Available: https://learning.oreilly.com/library/view/python-in-a/9781491913833/ch25.html. [Accessed: May 24, 2022]

[21] "The Dhall configuration language." Dhall [Online]. Available: https://dhall-lang.org/. [Accessed: Jan. 01, 2022]

[22] "CUE." Cue Language Community [Online]. Available: https://cuelang.org/. [Accessed: Jan. 01, 2022]

[23] "Jsonnet - The Data Templating Language." Code owned by Google [Online]. Available: https://jsonnet.org/. [Accessed: Jan. 01, 2022]

[24] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *The Site Reliability Workbook: Practical Ways to Implement SRE*. "O'Reilly Media, Inc.", 2018.

[25] "Nix." Nix/Nixpkgs/NixOS, May 2022 [Online]. Available: https://github.com/NixOS/nix. [Accessed: May 01, 2022]

[26] "NixOps." Nix/Nixpkgs/NixOS, May 2022 [Online]. Available: https://github.com/NixOS/nixops. [Accessed: May 24, 2022]

[27] "Gollum – A git-based Wiki." Gollum, Jan. 2022 [Online]. Available: https://github.com/gollum/gollum. [Accessed: Jan. 04, 2022]

[28] James Turnbull, *The Terraform Book*. 2016 [Online]. Available: https://learning.oreilly.com/library/view/the-terraform-book/9780988820258/. [Accessed: May 24, 2022]

[29] M. Taylor and S. Vargo, *Learning Chef*. 2014 [Online]. Available: https://learning.oreilly.com/library/view/learning-chef/9781491945087/. [Accessed: May 24, 2022]

[30] Unit 42, "Highlights From the Unit 42 Cloud Threat Report, 2H 2021," *Unit42*. Sep. 2021 [Online]. Available: https://unit42.paloaltonetworks.com/cloud-threat-report-2h-2021/. [Accessed: Jan. 04, 2022]

[31] V. Tzvetkov and J. M. Rinaudo, "Integrating AWS CloudFormation security tests with AWS Security Hub and AWS CodeBuild reports," *Amazon Web Services*. Sep. 2020 [Online]. Available: https://aws.amazon.com/blogs/security/integrating-aws-cloudformation-security-tests-with-aws-security-hub-and-aws-codebuild-reports/. [Accessed: Jan. 04, 2022]

[32] "Nickel." Tweag, May 2022 [Online]. Available: https://github.com/tweag/nickel. [Accessed: May 24, 2022]

[33] Y. Hamdaoui, "RFC-001 Overriding," Tweag, May 2022 [Online]. Available: https://github.com/tweag/nickel/blob/f0dc86bd35241ddcb8e9d8482bdc2724ac1b2b19/rfcs/001-overriding.md. [Accessed: May 24, 2022]

[34] J. G. Siek and W. Taha, "Gradual Typing for Functional Languages," p. 12, 2006.

[35] P. Wadler and R. B. Findler, "Well-Typed Programs Can't Be Blamed," in *Programming Languages and Systems*, vol. 5502, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–16 [Online]. Available: http://link.springer.com/10.1007/978-3-642-00590-9_1. [Accessed: May 24, 2022]

[36] J. des Rivieres and J. Wiegand, "Eclipse: A platform for integrating development tools. IBM Systems Journal, 43(2), 371-383," *IBM Systems Journal*, vol. 43, pp. 371–383, Jan. 2004, doi: 10.1147/sj.432.0371.

[37] E. Burnette, *Eclipse IDE: Pocket guide*, 1st ed. Sebastopol, CA: O'Reilly, 2005.

[38] "IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains," *JetBrains*. [Online]. Available: https://www.jetbrains.com/idea/. [Accessed: May 01, 2022]

[39] "Visual Studio: IDE and Code Editor for Software Developers and Teams." [Online]. Available: https://visualstudio.microsoft.com/. [Accessed: May 01, 2022]

[40] Moresmau, "EclipseFP The Haskell plug-in for Eclipse." [Online]. Available: https://eclipsefp.github.io/index.html. [Accessed: May 01, 2022]

[41] "IntelliJ-Haskell - IntelliJ IDEs Plugin | Marketplace," *JetBrains Marketplace*. [Online]. Available: https://plugins.jetbrains.com/plugin/8258-intellij-haskell. [Accessed: May 01, 2022]

[42] "HaskForce - The Haskell plugin for IntelliJ IDEA." [Online]. Available: https://haskforce.com/. [Accessed: Feb. 19, 2022]

[43] "Haskell for Mac IDE — Learn Functional Programming with Haskell." [Online]. Available: http://haskellformac.com/. [Accessed: Dec. 28, 2021]

[44] Y. Cébron, "Custom Language Support," *IntelliJ Platform Plugin SDK Help*. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/custom-language-support.html. [Accessed: Feb. 20, 2022]

[45] S. Keidel, W. Pfeiffer, and S. Erdweg, "The IDE portability problem and its solution in Monto," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, Oct. 2016, pp. 152–162, doi: 10.1145/2997364.2997368 [Online]. Available: https://doi.org/10.1145/2997364.2997368. [Accessed: Dec. 21, 2021]

[46] A. M. Sloane, M. Roberts, S. Buckley, and S. Muscat, "Monto: A Disintegrated Development Environment," in *Software Language Engineering*, 2014, pp. 211–220, doi: 10.1007/978-3-319-11245-9_12.

[47] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86, doi: 10.1109/CGO.2004.1281665.

[48] C. Lattner, "LLVM and Clang: Next Generation Compiler Technology," vol. BSDCan, p. 33, May 2008 [Online]. Available: https://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.pdf

[49] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, Oct. 2014, doi: 10.1145/2692956.2663188. [Online]. Available: https://doi.org/10.1145/2692956.2663188. [Accessed: May 01, 2022]

[50] NVIDIA, “CUDA Compiler Driver NVCC.” May 2022 [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf

[51] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. "O'Reilly Media, Inc.", 2013.

[52] OCaml Community, “OCaml - Merlin,” *An editor service that provides advanced IDE features for OCaml.* [Online]. Available: https://ocaml.github.io/merlin/. [Accessed: Feb. 20, 2022]

[53] F. Bour, T. Refis, and G. Scherer, “Merlin: A Language Server for OCaml (Experience Report),” *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 1–15, Jul. 2018, doi: 10.1145/3236798. [Online]. Available: http://arxiv.org/abs/1807.06702. [Accessed: Nov. 12, 2021]

[54] S. Vaughan-Nichols, “Protocol aims to be a programming standard,” *ZDNet*. [Online]. Available: https://www.zdnet.com/article/open-source-microsoft-protocol-aims-to-be-a-programming-standard/. [Accessed: Dec. 28, 2021]

[55] Red Hat, “Red Hat, Codenvy and Microsoft Collaborate on Language Server Protocol.” Jun. 2016 [Online]. Available: https://www.redhat.com/en/about/press-releases/red-hat-codenvy-and-microsoft-collaborate-language-server-protocol. [Accessed: May 22, 2022]

[56] Microsoft, “Language Servers.” [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/servers/. [Accessed: May 01, 2022]

[57] “Bringing a great IDE experience to the Rust programming language.” *rust-analyzer*. [Online]. Available: https://rust-analyzer.github.io/. [Accessed: Nov. 23, 2021]

[58] OCaml Community, “OCaml Language Server Protocol implementation.” OCaml, Apr. 2022 [Online]. Available: https://github.com/ocaml/ocaml-lsp. [Accessed: May 01, 2022]

[59] N. Mitchell *et al.*, “Building an Integrated Development Environment (IDE) on top of a Build System: The tale of a Haskell IDE,” in *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, Sep. 2020, pp. 1–10, doi: 10.1145/3462172.3462180 [Online]. Available: https://dl.acm.org/doi/10.1145/3462172.3462180. [Accessed: Nov. 12, 2021]

[60] I. Wechsung, “The Frege Programming Language (Draft),” p. 124 [Online]. Available: http://web.mit.edu/frege-lang_v3.24/Language.pdf

[61] T. Gagnaux and D. Konig, “Developing a Minimal Language Server for the Frege Programming Language: An Experience Report,” p. 7.

[62] E. Dolstra, M. de Jonge, and E. Visser, “Nix: A Safe and Policy-Free System for Software Deployment,” p. 14, 2004.

[63] “Rnix-lsp.” Nix community projects, Apr. 2022 [Online]. Available: https://github.com/nix-community/rnix-lsp. [Accessed: May 01, 2022]

[64] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," *arXiv:0902.0019 [cs]*, Jan. 2009 [Online]. Available: http://arxiv.org/abs/0902.0019. [Accessed: Feb. 24, 2022]

[65] A. Leimeister, "A Language Server and IDE Plugin for CPAchecker," Master's thesis, Ludwig-Maximilians-Universität München, 2020 [Online]. Available: https://www.sosy-lab.org/research/bsc/2020.Leimeister.A_Language_Server_and_IDE_Plugin_for_CPAchecker.pdf

[66] Z. Porkoláb, T. Brunner, D. Krupp, and M. Csordás, "Codecompass: An open software comprehension framework for industrial usage," in *Proceedings of the 26th Conference on Program Comprehension*, May 2018, pp. 361–369, doi: 10.1145/3196321.3197546 [Online]. Available: https://doi.org/10.1145/3196321.3197546. [Accessed: Feb. 25, 2022]

[67] H. Bünder and H. Kuchen, "Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol," in *Model-Driven Engineering and Software Development*, 2020, pp. 225–245, doi: 10.1007/978-3-030-37873-8_10.

[68] "Rnix-parser." Nix community projects, May 2022 [Online]. Available: https://github.com/nix-community/rnix-parser. [Accessed: May 01, 2022]

[69] D. Beyer, G. Dresler, and P. Wendler, "Software Verification in the Google App-Engine Cloud," in *Computer Aided Verification*, vol. 8559, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, A. Biere, and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 327–333 [Online]. Available: http://link.springer.com/10.1007/978-3-319-08867-9_21. [Accessed: Feb. 24, 2022]

[70] D. Beyer, "CPAchecker In the VerifierCloud." Software Systems Lab LMU [Online]. Available: https://vcloud.sosy-lab.org/cpachecker/webclient/help/. [Accessed: Feb. 24, 2022]

[71] M. Mészáros, M. Cserép, and A. Fekete, "Delivering comprehension features into source code editors through LSP," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2019, pp. 1581–1586, doi: 10.23919/MIPRO.2019.8756695.

[72] L. Bettini and S. Efftinge, *Implementing domain-specific languages with Xtext and Xtend: Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*, Second edition. Birmingham Mumbai: Packt Publishing, 2016.

[73] L. C. L. Kats and E. Visser, "The spoofax language workbench: Rules for declarative specification of languages and IDEs," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, Oct. 2010, pp. 444–463, doi: 10.1145/1869459.1869497 [Online]. Available: https://doi.org/10.1145/1869459.1869497. [Accessed: Dec. 21, 2021]

[74] V. Pech, "JetBrains MPS: Why Modern Language Workbenches Matter," in *Domain-Specific Languages in Practice: With JetBrains MPS*, A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, Eds. Cham: Springer International Publishing, 2021, pp. 1–22 [Online]. Available: https://doi.org/10.1007/978-3-030-73758-0_1. [Accessed: May 08, 2022]

[75] "Monaco Editor." Microsoft, May 2022 [Online]. Available: https://github.com/microsoft/monaco-editor. [Accessed: May 01, 2022]

[76] L. Vogel and M. Milinkovich, "Eclipse rich client platform. Vogella series, lars vogel (2015)."

[77] R. Rodriguez-Echeverria, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, "Towards a Language Server Protocol Infrastructure for Graphical Modeling," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Oct. 2018, pp. 370–380, doi: 10.1145/3239372.3239383 [Online]. Available: https://doi.org/10.1145/3239372.3239383. [Accessed: Mar. 01, 2022]

[78] J. K. Rask, F. P. Madsen, N. Battle, H. D. Macedo, and P. G. Larsen, "The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions," *Electronic Proceedings in Theoretical Computer Science*, vol. 338, pp. 3–18, Aug. 2021, doi: 10.4204/EPTCS.338.3. [Online]. Available: http://arxiv.org/abs/2108.02961. [Accessed: Feb. 15, 2022]

[79] J. Rask and F. Madsen, "Decoupling of Core Analysis Support for Specification Languages from User Interfaces in Integrated Development Environments," Master's thesis, Aarhus University, 2021.

[80] Scala Center and Jetbrains, "Build Server Protocol · Protocol for IDEs and build tools to communicate about compile, run, test, debug and more." [Online]. Available: https://build-server-protocol.github.io/. [Accessed: May 01, 2022]

[81] "Debug Adapter Protocol," *Official page for Debug Adapter Protocol.* [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/implementors/adapters/. [Accessed: Mar. 03, 2022]

[82] P. Jeanjean, B. Combemale, and O. Barais, "IDE as Code: Reifying Language Protocols as First-Class Citizens," in *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, Feb. 2021, pp. 1–5, doi: 10.1145/3452383.3452406 [Online]. Available: https://dl.acm.org/doi/10.1145/3452383.3452406. [Accessed: Mar. 03, 2022]