Contents

Chapter 1

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. This chapter will survey a varied range of popular language servers, compare common capabilities, and implementation approaches. Additionally, this part aims to recognize alternative approaches to the LSP, in the form of legacy protocols, extensible development platforms LSP extensions and the emerging Language Server Index Format.

1.1 Previous Approaches

1.1.1 IDEs

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 (**intellij-supported-languages?**). 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(**eclipse-a-platform?**,eclipse-www), IntelliJ(**intelliJ?**), and Visual Studio(**VisualStudio?**), 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 ((**jetbrains-all-products?**)). 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 (**haskell-ide-eclips?**), as well as independent IntelliJ plugins (**HaskForce?**). 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(**haskell-for-mac?**) 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.

In general, developing language integrations, both as the vendor of an IDE or a third-party plugin developer requires extensive resources. Table 1.1 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 (**custom-language-support?**), 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 1.1: 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 (**intellij-comparison-eclipse?**). Editor-specific approaches also tend to lock-in programmers into a specific platform for its language support regardless of their personal preference.

1.1.2 IDE Abstraction

1.1.2.1 Monto

The authors of the Monto project(**monto-disintegrated?**) 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 (**llvm?**). Compiler frontends for different languages – e.g. Clang(**clang?**), Rustc(**rustc?**), NVCC(**nvcc?**),... – 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[zeromq] technology which was chosen because it is lightweight and available in manly languages (**monto-disintegrated?**) allowing to make use of existing language tools.

1.1.2.2 Merlin

The Merlin tool (**merlin-website?**) 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 (**merlin?**).

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.

1.2 Language Servers

The LSP project was announced (**lsp-announced?**) 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` (**Spec?**). Microsoft's official website lists 172 implementations of the LSP(**implementations?**) 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.

1.2.1 Considerable dimensions

To be able to compare and describe LSP projects objectively and comprehensively, the focus will be on the following dimensions.

Target Language The complexity of implementing language servers is influenced severely by the targeted language. Feature rich languages naturally require more sophisticated solutions. Yet, existing tooling can often be leveraged to facilitate language servers.

Features The LSP defines an extensive array of capabilities. The implementation of these protocol features is optional and servers and clients are able to communicate a set of *mutually supported* capabilities.

The Langserver (**langserver.org?**) project identified six basic capabilities that are most widely supported:

1. Code completion,
2. Hover information,
3. Jump to definition,
4. Find references,
5. Workspace symbols,
6. Diagnostics

Yet, not all of these are applicable in every case and some LSP implementations reach for a much more complete coverage of the protocol.

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.

1.2.2 Representative LSP Projects

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* (**rust-analyzer?**), *ocaml-lsp/merlin* (**merlin?**) and the *Haskell Language Server* (**hls?**)
2. A project that indirectly interacts with the language implementation through an interactive programming shell (REPL). *Frege LSP (**frege-lsp?**)*
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* (**rnix-lsp?**)
4. Two projects that facilitate the LSP as an interface to an existing tool via HTTP or command line. *CPAchecker*(**cpachecker-lsp?**) and *CodeCompass*(**comprehension-features?**)
5. An approach to generate language servers from domain specific language specifications (**multi-editor-support?**).

1.2.2.1 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.

1.2.2.1.1 HLS For instance the Haskell language server facilitates a plugin system that allows it to integrate with existing tooling projects (**hls-plugin-search?**). Plugins provide capabilities for linting (**hls-hlint-plugin?**), formatting (**hls-floskell-plugin?**,hls-ormolu-plugin), documentation (**hls-haddock-comments-plugin?**) and other code actions (**hls-tactics-plugin?**) 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.

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

1.2.2.1.3 Rust-Analyzer The rust-analyzer (**rust-analyzer?**) 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.

1.2.2.1.4 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 (**frege-github?**).

Frege as introduced in (**frege-paper?**) 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 (**frege-eclipse?**), the tooling efforts have since been put towards an LSP implementation. The initial development of this language server has been reported on in (**frege-lsp-report?**). 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.

1.2.2.2 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(**rnix-git?**) language server for the Nix(**nixos.org?**) programming language. Despite the Nix language being written in C++ (**nix-repo?**), its language server builds on a custom parser called "rnix" (**rnix?**) 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.

1.2.2.3 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.

1.2.2.3.1 CPAchecker The work presented by Leimeister in (**cpachecker-lsp?**) 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(**cpachecker?**). 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 (**cpa-clusters?**). 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.

1.2.2.3.2 CodeCompass Similar to the work by Leimeister (c.f. sec. 1.2.2.3.1), in (**comprehension-features?**) 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 (**code-compass?**). 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.

1.2.2.4 Language Servers generation for Domain Specific Languages

Bünder and Kuchen (**multi-editor-support?**) 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 (**eclipse-xtext?**), Spoofax (**spoofax?**) or MPS(**jetpbrains-mps?**) 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 (**multi-editor-support?**) present how Xtext can generate an LSP server for a custom DSL, providing multi-editor support. The authors especially mention the Monaco Editor (**monaco-editor?**), a reusable HTML component for code editing using web technologies. It is used in products like VSCode (**vscode?**), Theia (**theia?**) 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.

1.2.3 Honorable mentions

1.3 Alternative approaches

1.3.1 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. In sec. **??** 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 (**rust-analyzer-extensions?**), 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 (**lsp-for-graphical-modeling?**). 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 (**decoupling-core-analysis-support?**) 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 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.

1.3.2 Language Server Index Format

Nowadays, code hosting platforms are an integral part of the developer toolset (GitHub(**github?**), Sourcegraph(**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 in sec. **??** for the interface declaration in lst. 1.1 can be represented using the LSIF. Figure 1.1 visualizes the

result (cf. lst. 1.2). 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 1.1 Exemplary code snippet to showing LSIF formatting

```
export interface ResultSet {
}
```

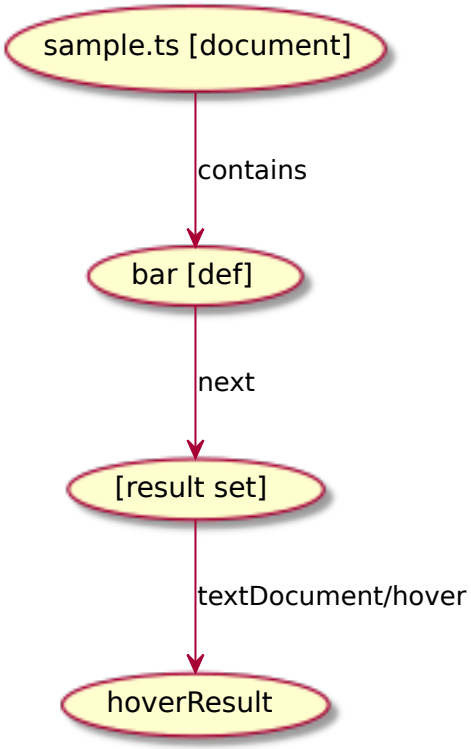

Figure 1.1: LSIF encoded graph for the exemplary code

Listing 1.2 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.

1.3.3 *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 in sec. 1.3.1, 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 (**build-server-protocol?**), 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(**DAP-impls?**).

Following the named protocols, Jeanjean et al. envision a future (**reifying?**) 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.