# 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 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.

### 1.1.1 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.

###### 1.1.1.1 LSP compliance

###### 1.1.1.2 Features

###### 1.1.1.3 File processing

###### 1.1.1.3.1 Incremental

###### 1.1.1.3.2 Full

### 1.1.2 Comparative Projects

### 1.1.3 Honorable mentions

## 1.2 Alternative approaches

### 1.2.1 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.

### 1.2.2 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 LSIF data provided in one shot by the server.

The LSIF specification (**lsif-spec?**) defines four principal goals:

- The format should not imply the use of a certain persistence technology.
- The data defined should be modeled as closely as possible to the Language Server Protocol to make it possible to serve the data through the LSP without further transformation.
- The data stored is result data usually returned from an LSP request.
- The output format will be based on JSON as with the LSP.

The LSIF format is a graph structure that links source code spans to language analysis results. The graph structure mimics LSP types. Vertices represent higher level concepts such as `documents`, `ranges`, `resultSets` and actual results. The relation between vertices is expressed through the edges.

Referring to an example form the official specification (**lsif-spec?**), an analysis of the code sample in lst. **??** may produce hover information for the function `bar()`. Using the LSIF, the result would be encoded as seen in lst. **??**. The graph structure encoded here is visualized in fig. **??**. 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

As a consequence, a subset of LSP capabilities can be provided statically based on the preprocessed LSIF model.

### 1.2.3 *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. **??**, 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.