Chapter 1

Evaluation

Section **??** described the implementation of the Nickel Language Server addressing the first research question stated in sec. **??**. Proving the viability of the result and answering the second research question demands an evaluation of different factors.

Earlier, the most important metrics of interest were identified as:

Usability What is the real-world value of the language server?
Does it improve the experience of developers using Nickel? NLS offers several features, that are intended to help developers using the language. The evaluation should assess whether developers experience any help due to the use of the server.
Does NLS meet its users' expectations in terms of completeness and correctness and behavior? Being marketed as a Language Server, invokes certain expectations due to previous experience with other languages and language servers. Here, the evaluation should show whether NLS lives up to the expectations of its users.

Performance What are the typical latencies of standard tasks? In this context *latency* refers to the time it takes from issuing an LSP command to return of the reply by the server. The JSON-RPC protocol used by the LSP is synchronous, i.e. requires the server to return results of commands in the order it received them. Since most commands are sent implicitly, a quick processing is imperative to avoid commands queuing up.
Can single performance bottlenecks be identified? Single commands with excessive runtimes can slow down the entire communication resulting in bad user experience. Identified issues can guide the future work on the server.
How does the performance of NLS scale for bigger projects? With increasing project sizes the work required to process files increases as well. The evaluation should allow estimates of the sustained performance in real-world scenarios.

Answering the questions above, this chapter consists of two main sections. The first section sec. **??** introduces methods employed for the evaluation. In particular, it details the survey (sec. **??**) which was conducted with the intent

to gain qualitative opinions by users, as well as the tracing mechanism (sec. **??**) for factual quantitative insights. Section **??** summarises the results of these methods.

1.1 Methods

1.1.1 Qualitative

1.1.2 Quantitative

1.2 Results

1.2.1 Process

1.2.2 Qualitative

As outlined in [#sec:qualitative-study-outline], the qualitative study consists of two parts conducted before and after an introductory workshop. The pre-evaluation aimed to catch the users's expected features and behaviours, while the main survey asked users about their concrete experiences with the NLS.

1.2.2.1 Pre-Evaluation

Responding to the first point (c.f. [#sec:expected-features]), the participants unanimously identified four of the six foundational language server capabilities that guided the implementation of the project (c.f. sec. **??**);

Type-information on hover was named almost uniformly. The participants showed a special interest in this feature describing specific behaviours. The desired information exposed by this feature are value types including applied contracts and documentation as well as function signatures.
When asked about the hover LSP method in particular, participants name additional function documentation, default values and the visualization of scopes as an additional features.

Diagnostics are widely understood as an important feature. Participant had very particular opinions about the behavior and detail of diagnostics including error message at the correct location in the code signaling syntax errors or possibly evaluation errors and contract breaches. In either case the diagnostic should be produced "On-the-fly" while typing or upon saving the document.
When asked about the diagnostics feature of language servers directly, the answers corroborated these initial opinions. In addition some participants named code linting, i.e. warnings about code style, unused variables, deprecated code and undocumented elements, as well as structural analysis hints as possible features. Structural analysis was imagined to go that far as being able to "suggest how to fix" mistakes in the code.

Code Completion was equally name in all but one response. It was described as a way to chose from possible completion candidates of options. The answers included aspirational vague descriptions of such a feature including the a way to automatically prioritize specific items.

> Responding about the concrete LSP feature, participants listed variables, record fields, types, functions and function argument candiates as possible completion candidates. Moreover, some suggested the inclusion of the completion context to guide priorization as well as auto-generated contract and function skeletons.

Jump-to-Definition ~ was included in three fourth of responses. ~ The specific feature survey revealed the exoected behaviour in more detail; In general, the participants expect the feature to work with any kind of reference, i.e., variable usages, function calls, function arguments and type annotations. Record fields ale equally desired although the ability to define self referencing records was pointed out as a challenge. However, subjects expect statically defined nested fields to point to the correct respective definition.

The other two features Find-References and Workspace/Document Symbols on the contrary were sparingly commented. Participands noted that they did not use these capabilities. The features were however well understood, as shown by some responsess naming very particular distinctions of symbol types.

Beyond features that were explicitly targeted by this work, syntax highlighting and code formatting as well as error tolerance were named as further desireable features of a language server. Error tolerance was detailed as the capability of the language server to continue processing and delivering analysis of invalid sources restricting the computation to the correct parts of the program.

1.2.2.2 Experience Survey

The above figures show the turnout of three items from the survey for each of the relevant features. Neither of them shows clear trends with positive and negative reults distributed almost evenly between positive and negative sentiments.

The first graph (fig. **??**) represents the participants' general experience with the relevant features. It shows that each feature worked without issue in at least one instance. Yet, three features were reported to not work at all and no feature left the users unsurprised. Participant found the hover and diagnostic features to behave particularly unexpectedly.

For the second item of each feature, the survey asked the subjects to rate the quality of the language server based on their expectations. Figure **??** summarizes the results. Apart from the same three occasions in which a feature did not work for one participant, the majority of responses show that NLS met its user's expectations at least partially. The results are however highly polarized as the Jump-to-Definition and Hover features demonstrate; Each received equally many votes for being inapt and fully able to hold up to the participants expectations at the same time. Other features were left with with a nuniformly distributed assessment (e.g. Completion and Find-References). The clearest result was achieved by the Diagnostics feature, which received a slight but uncontended positive sentiment.

Asking about the general satisfaction with each feature, results in the same mixed answers as seen in fig. **??**. While a slight majority of responses falls into the upper half of the possible spectrum, two features (of the three that have previously been reported without function) were given the lowest possible rating.

1.2.2.3 Hover

As apparent in (fig. **??**), most participants experienced unexpected behavior by the LSP when using the hover functionality. In the comments, extraneous debug output and incorrect displaying of the output by the IDE are pointed out as concrete examples. However one answer suggests that the feature was working with "usually useful" output.

1.2.2.4 Diagnostics

While the diagnostics shown by NLS appear to behave unexpectedly for some users in fig. **??**, all participants marked that those did not deter from keep using NLS for it as displayed in fig. **??**. In the comments some respondents praised the "quick" and "direct feedback" as well as the visual error markers pointing to the exact locations of possible issues while others mentioned "unclear messages." However, it was pointed out that it contracts were not checked by the Language Server. Moreover, a performance issue was brought up noting that in some situations NLS "queues a lot of work and does not respond."

1.2.2.5 Code Completion

Comments about the Code Completion feature were unanimously critical. Some participants noted the little gained "value over the token based completion built into the editor" while others specifically pointed at "missing type information and docs." Additionally record field completion was found to be missing, yet highly valued.

1.2.2.6 Document Navigation

Results and comments about the Go-To-Definition and Find-Referneces were polarized. On the one hand users reported no issues while others experienced unexpected behavior or were unabke to use the feature at all (cf. fig. **??**). Similarly, the comments on one hand suggest that "the feature works well and is quick" while on the other mention inconsistencies and unavailabilty. More practically, cross file navigation was named an important missing feature.

1.2.2.7 General Performance

The responses to this item suggest that NLS' perfomance is largely dependent on its usage. On unmodified files queries were reported to evaluate "instantaniously." However modifying files caused that "modifications stack up" causing high CPU usage and generally "very slow" responses. Others pointed out that documentation was slow to resolve while the server itself was "generally fast."

1.2.3 Quantitative

The quantitative evaluation focuses on the performance characteristics of NLS. As described in sec. **??** a tracing module was embedded into the NLS binary which recorded the runtime together with the size of the analyzed data, i.e., the number of linearization items sec. **??** or size of the analyzed file.

1.2.3.1 Dataset

The underlying data set consists of 16760 unique trace records. Since the `textDocument/didOpen` method is executed on every update of the source, it greatly outnumbers the other events. The final distrubution of methods traced is:

Table 1.1: Number of traces per LSP method

Method	count	linearization based
`textDocument/didOpen`	13436	no
`textDocument/completion`	2981	yes
`textDocument/hover`	227	yes
`textDocument/definition`	68	yes
`textDocument/references`	49	yes

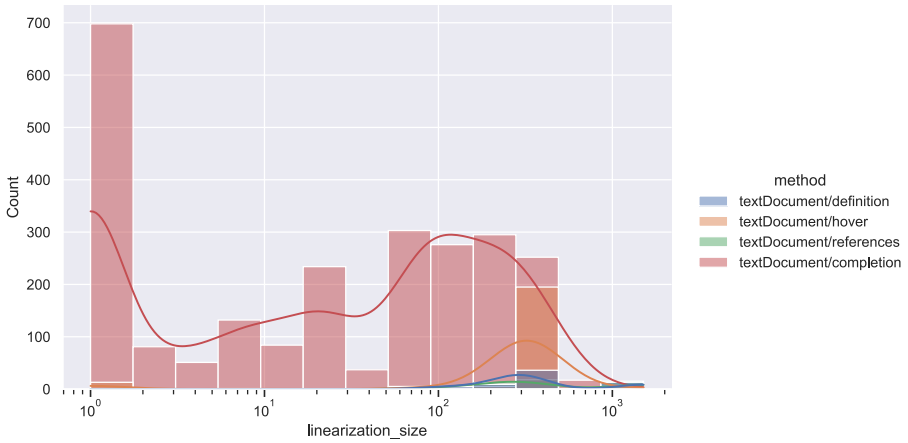


Figure 1.1: Distribution of linearization based LSP requests

Figures 1.1 break up these numbers by method and linearization size or file size respectively. The first figure shows a peak number of traces for completion events between 0 to 1 linearization items as well as local maxima around a linearization size of 20 to 30 and sustained usage of completion requests in files of $90 - 400$ items. Similar to the completion requests (but well outnumbered in total counts), other other methods were used mainly in the range between 200 and 400 linearization items. A visualization of the Empirical Cumulative Distribution Function (ECFD) (fig. **??** corroborates these findings. Moreover, it shows an additional hike of Jump-to-Definition and Find-References calls at on files with around 1500 linearization items. The findings for linearization based methods line up with those depicting linearization events (identified as `textDocument/didOpen`). An initial peak referring to rather small input files between 300 and 400 bytes in size is followed by a sustained usage of the NLS on files with 2 to 6 kiloBytes of content topped with a final application on 35 kiloByte large data.

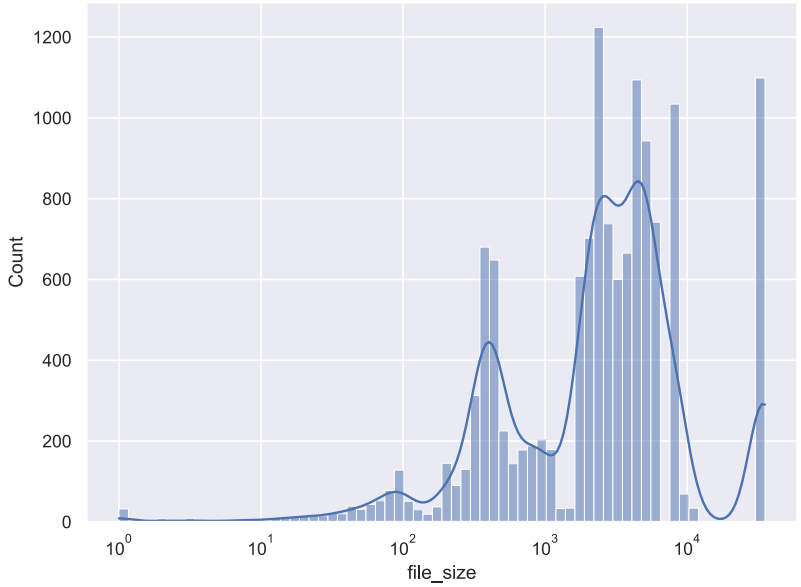


Figure 1.2: Distribution of file analysis requests

1.2.3.2 Big Picture Latencies

Comparing the runtime of the individual methods alone in fig. 1.3, reveals three key findings. First, all linearization based methods exhibit a sub-millisecond latency in at least 95 of all invocations and median response times fo less than $100ţs$ However, maximum latencies of completion invocations reached tens of milliseconds and in one recorded case about $300ms$. Finally, document linearization as associated with the `textDocument/didOpen` method shows a great range with maxima of $1.5 * 10^5 ţs$ (about 2.5 minutes) and a generally greater inter quartile range spanning more than tow orders of magnitude.

1.2.3.3 Special cases

Setting the runtime of completion requests in relation to the linearization size on which the command was performed, shows no clear correlation between the dimensions. In fact the correlation coefficient between both variables measures 0.01617 on a linear scale and 0.26 on a $\log_{10} \log_{10}$ scale. Instead, vertical colums stand out in the correlation graph fig. 1.4a. The height of these colums varies from one to five orders of magnitude. Considering the item density shows that especially high columns form whenever the server receives a higher load of requests. Additionally color coding the individual requests by time reveils that the trace points of each column were recorded at a short time interval. Applying the same analysis to the other methods in figs. 1.4b, 1.4c, **??** returns similar findings, although the columns remain more compact in comparison to the Completions method. In case of the `didOpen` method columns are clearly

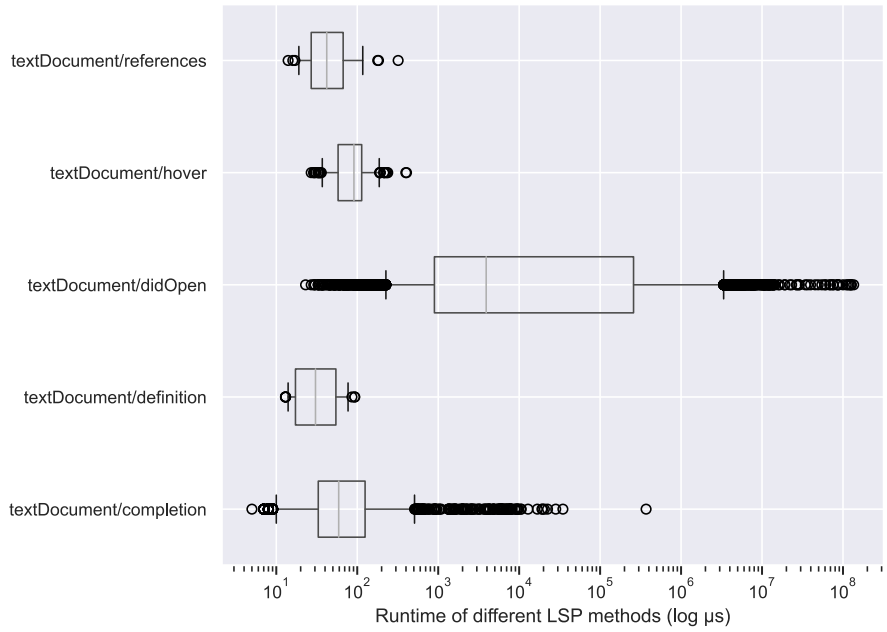
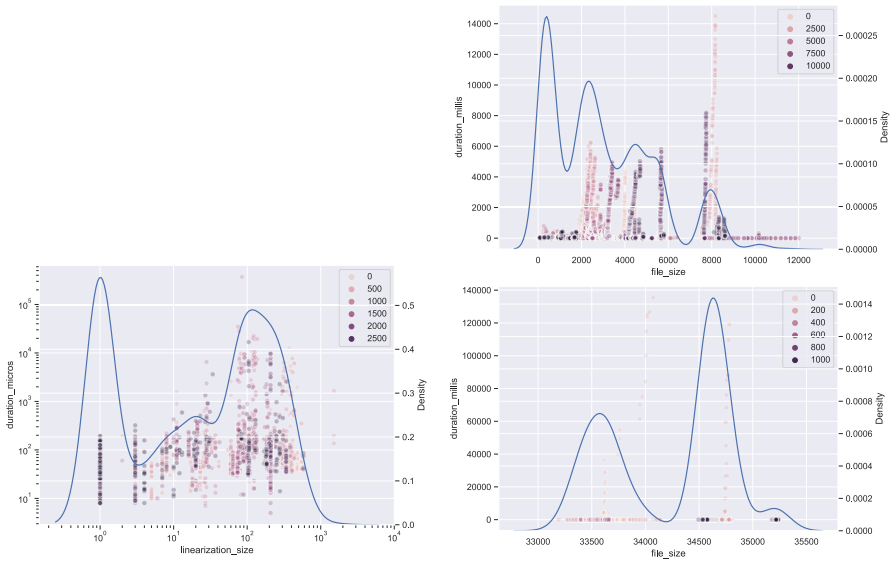


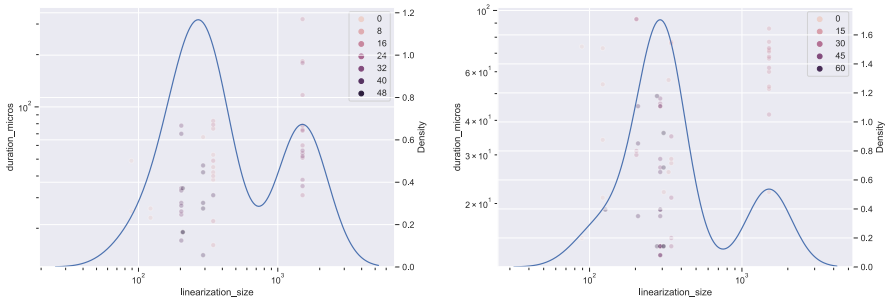
Figure 1.3: Statistical runtime of different LSP methods

visible too. However, here they appear leaning as suggesting an increase in computation time as the file grows during a single series of changes to the file.



(a) Runtime latencies of completion requests at differnet linearization sizes

(b) Runtime latencies of hover requests at differnet linearization sizes



(c) Runtime latencies of find-references requests at differnet linearization sizes

(d) Runtime latencies of jump-to-definition requests at differnet linearization sizes

Figure 1.4: Runtime latencies of different linearization based methods

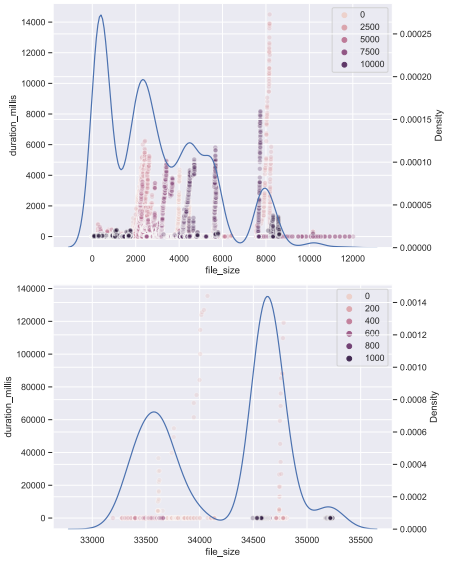


Figure 1.5: Runtime latencies of file update handlings at different file sizes