Contents

1 Evaluation **3**
1.1 Evaluation Considerations 4
1.2 Methods 4
1.2.1 Objectives 4
1.2.2 Qualitative Evaluation Setup {#sec:qualitative@methods} 5
1.2.3 Quantitative {#sec:quantitative@methods} 6
1.3 Results 7
1.3.1 Qualitative 7
1.3.2 Quantitative 10
1.4 Discussion 14
1.4.1 Diagnostics 14
1.4.2 Cross File Navigation 15
1.4.3 Autocompletion 15
1.4.4 Performance 16

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 the server does improve the experience of developers.
Does NLS meet its users' expectations in terms of completeness, correctness and behavior? Labeling NLS as a Language Server, invokes certain expectations built up by 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 the reply of 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. 1.2 introduces methods employed for the evaluation. In particular, it details the survey (sec. 1.3.1) which was conducted with the intent

to gain qualitative opinions by users, as well as the tracing mechanism (sec. 1.3.2) for factual quantitative insights. Section 1.3 summarizes the results of these methods.

1.1 Evaluation Considerations

Different methods to evaluate the abovementioned metrics were considered. While quantifying user experience yields statistically sound insights about the studied subject, it fails to point out specific user needs. Therefore, this work employs a more subjective evaluation based on a standardized experience report focusing on individual features. Contrasting the expectations highlights well executed, immature or missing features. This allows more actionable planning of the future development to meet user expectations.

On the other hand it is more approachable to track runtime performance objectively through time measurements. In fact, runtime behavior was a central assumption underlying the server architecture. As discussed in sec. **??** NLS follows an eager, non-incremental processing model. While incremental implementations are more efficient, as they do not require entire files to be updated, they require explicit language support, i.e., an incremental parser and analysis. Implementing these functions exceeds the scope of this work. Choosing a non-incremental model on the other hand allowed reusing entire modules of the Nickel language. The analysis itself can be implemented both in a lazy or eager fashion. Lazy analysis implies that the majority of information is resolved only upon request instead of ahead of time. That is, an LSP request is delayed by the analysis before a response is made. Some lazy models also support memorizing requests, avoiding recomputing previously requested values. However, eager approaches preprocess the file ahead of time and store the analysis results such that requests can be handled mostly through value lookups. To fit Nickels' type-checking model and considering that in a typical Nickel workflow, the analysis should still be reasonably efficient, the eager processing model was chosen over a lazy one.

1.2 Methods

1.2.1 Objectives

The qualitative evaluation was conducted with a strong focus on the first metric in [sec:metrics]. Usability proves hard to quantify, as it is tightly connected to subjective perception, expectations and tolerances. The structure of the survey is guided by two additional objectives, endorsing the separation of individual features. On one hand, the survey should inform the future development of NLS; which feature has to be improved, which bugs exist, what do users expect. This data is important for NLS both as an LSP implementation for Nickel (affecting the perceived maturity of Nickel) and a generic basis for other projects. On the other hand, since all features are essentially queries to the common linearization data structure (cf. sec. **??**), the implementation of this central structure is an essential consideration. The survey should therefore also uncover apparent problems with this architecture. This entails the use of language abstractions (cf.

sec. **??**) and the integration of Nickel core functions such as the type checking procedure.

The quantitative study in contrast focuses on measurable performance. Similarly to the survey-based evaluation, the quantitative study should reveal insight for different features and tasks separately. The focus lies on uncovering potential spikes in latencies, and making empirical observations about the influence of Nickel file sizes.

1.2.2 Qualitative Evaluation Setup {#sec:qualitative@methods}

Inspired by the work of Leimeister in (**leimeister?**), a survey aims to provide practical insights into the experience of future users. In order to get a clear picture of the users' needs and expectations independently of the experience, the survey consists of two parts – a pre-evaluation and final survey.

1.2.2.1 Pre-Evaluation

1.2.2.1.1 Expected features The pre-evaluation introduced participants in brief to the concept of language servers and asked them to write down their understanding of several LSP features. In total, six features were surveyed corresponding to the implementation as outlined in sec. **??**, namely:

1.2.2.1.2 Expected behavior

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 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. The item for the "Hover" feature for instance reads as follows:

   > Editors can show some additional information about code under the cursor. The selection, kind, and formatting of that information is left to the Language Server.
   >
   > What kind of information do you expect to see when hovering code? Does the position or kind of element matter? If so, how?

Items first introduce a feature on a high level followed by asking the participant to describe their ideal implementation of the feature.

1.2.2.2 Experience Survey

For the final survey, interested participants at Tweag were invited to a workshop introducing Nickel. The workshop allowed participants unfamiliar with the Nickel language to use the language and experience NLS in a more natural setting.

Following the workshop, participants filled in a second survey which focused on three main aspects:

First, the general experience of every individual feature. Without weighing their in expectations, the participants were asked to give a short statement of their experience. The item consists of a loose list of statements with the aim to achieve a rough quality classification:

☐ The feature did not work at all
☐ The feature behaved unexpectedly
☐ The feature did not work in all cases
☐ The feature worked without an issue
☐ Other

The following items survey the perceived performance and stability. The items were implemented as linear scales that span from "Very slow response" to "Very quick response" and "Never Crashed" to "Always Crashed" respectively. The second category asked participants to explicitly reflect on their expectations:

○ The feature did not work at all
○ Little of my expectation was met
○ Some expectations were met, enough to keep using NLS for this feature
○ Most to all expectations were met ○ NLS surpassed the expectations
○ Other

In the final part participants could elaborate on their answers.

Why were they (not) satisfied?
What is missing, what did they not expect?

1.2.3 Quantitative {#sec:quantitative@methods}

To address the performance metrics introduced in sec. **??**, a quantitative study was conducted, that analyzes latencies in the LSP-Server-Client communication. The study complements the subjective reports collected through the survey (cf. sec. 1.2.2.2). The evaluation is possible due to the inclusion of a custom tracing module in NLS. The tracing module is used to create a report for every request, containing the processing time and a measure of the size of the analyzed document. If enabled, NLS records an incoming request with an identifier and time stamp. While processing the request, it adds additional data to the record, i.e., the type of request, the size of the linearization (cf. sec. **??**) or processed file and possible errors that occurred during the process. Once the server replies to a request, it records the total response time and writes the entire record to an external file.

The tracing approach narrows the focus of the performance evaluation to the time spent by NLS. Consequently, the performance evaluation is independent of the LSP client (editor) that is used. Unlike differences in hardware which affects all operations similarly, LSP clients may implement different behaviors that may cause editor-specific biases. For instance, the LSP does not specify the frequency at which file changes are detected, which in turn can lead to request queuing depending on the editor used.

1.3 Results

1.3.1 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' expected features and behaviors, while the main survey asked users about their concrete experiences with the NLS.

1.3.1.1 Pre-Evaluation

In the initial free assessment of expected features (c.f. [#sec:expected-features]) the participants unanimously identified four of the six language server capabilities that guided the implementation of the project (c.f. sec. **??**): Type-information on hover, automatic diagnostics, Code Completion and Jump-to-Definition.

The other two features, Find-References and Workspace/Document Symbols on the contrary were sparingly commented. Some participants noted that they did not use these capabilities.

1.3.1.1.1 Type-information on hover Hovering is expected to work on values as well as functions. For values, it is desired to show types including applied contracts, documentation and default values. On functions, it should display the function's signature and documentation. Additionally, hovering an item desirably visualizes the scope of the item, i.e. where it is available.

1.3.1.1.2 Diagnostics Diagnostics are expected to include error messages signalling syntax and type errors as well as possibly evaluation errors and contract breaches. The diagnostics should show up at the correct positions in the code and "suggest how to fix" mistakes. Code linting was named as a possible extension to error reporting. This would include warnings about bad code style – formatting, casing conventions – unused variables, deprecated code and undocumented elements. Moreover, structural analysis was conceived to allow finding structural issues and help to fix them In either case the diagnostic should be produced "On-the-fly" while typing or upon saving the document.

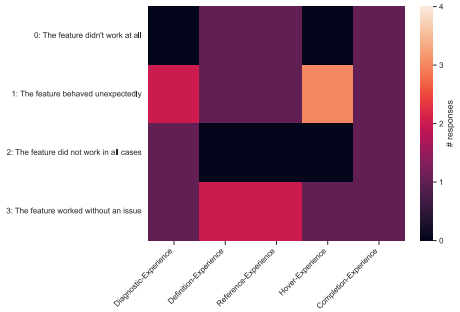
1.3.1.1.3 Code Completion Code Completion was described as a way to chose from possible completion candidates of options. Completable items can be variable names, record fields, types or functions. Besides, Participants conceived filtering or prioritizing of candidates by type if applied as function arguments. Finally, the completion context could guide prioritization as well as auto-generation of contract and function skeletons.

1.3.1.1.4 Jump-to-Definition Users expect Jump-to-Definition to work with any kind of reference i.e., variable usages, function calls, function arguments and type annotations. On records and references to records, users expect statically defined nested fields to point to the correct respective definition. The ability to define self referencing records was however conceded to be a challenge.

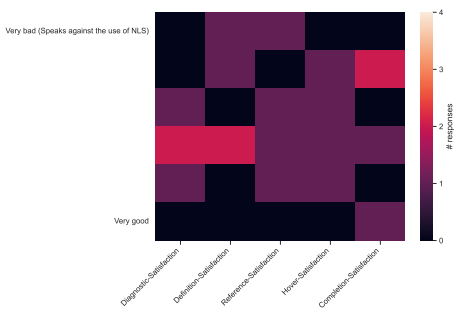
1.3.1.1.5 Other features Syntax highlighting and code formatting as well as error tolerance were named as further desirable features of a language server beyond the explicitly targeted features. Error tolerance was detailed as the capability of the language server to continue processing and delivering analysis of invalid sources. For invalid files a language server should still be able to provide its functionality for the correct parts of the program.

1.3.1.2 Experience Survey

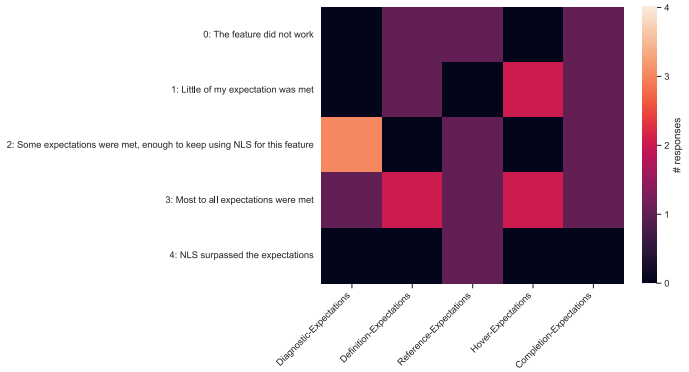
This subsection describes the results from the filled after the Nickel workshop in which participants were asked to install the LSP to support their experience. It first looks at a summary of the data, before diving into the comments for each directly addressed feature.



(a) Number of votes for each experience category and feature



(b) Number of votes for each satisfaction category and feature



(c) Number of votes for each expectation category and feature

Figure 1.1: User responses regarding general experience, fulfillment of expectations and general satisfaction.

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 results distributed almost evenly between positive and negative sentiment.

The first graph (fig. 1.1a) 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. Users found the hover and diagnostic features to behave particularly unexpectedly.

In the second item of each feature, the survey asked the subjects to rate the quality of the language server based on their expectations. Figure 1.1c summarizes the results. In agreement with the first graph, one user was unable to use at least three features entirely. 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 a uniformly distributed assessment (e.g. Completion and Find-References). The clearest result was achieved by the Diagnostics feature, which received a slight but uncontested positive sentiment.

The general satisfaction with each feature was answered in the same polarized manner as seen in fig. 1.1b. A slight majority of responses falls into the upper half of the possible spectrum. Two of the features reported without function in the preceding questions were given the lowest possible rating.

1.3.1.2.1 Hover {#sec:hover@res} As apparent in (fig. 1.1a), 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.

1.3.1.2.2 Diagnostics {#sec:diagnostics@res} While the diagnostics shown by NLS appear to behave unexpectedly for some users in fig. 1.1a, no user felt deterred from keep using NLS for it as displayed in fig. 1.1c. Some respondents praised the "quick" and "direct feedback" as well as the visual error markers pointing to the exact locations of possible issues. On the contrary, others mentioned "unclear messages" and pointed out that 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."

1.3.1.2.3 Code Completion {#sec:code-completion@res} 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, albeit highly valued.

1.3.1.2.4 Document Navigation {#sec:document-navigation@res} Results and comments about the Go-To-Definition and Find-References were polarized. Some users experienced unexpected behavior or were unable to use the feature at all (cf. fig. 1.1a). Similarly, the comments on one hand suggest that "the feature works well and is quick" while on the other mention inconsistencies and unavailability. More specifically, cross file navigation was named an important missing feature.

1.3.1.2.5 General Performance {#sec:general-performance@res} The responses to the general performance suggest that NLS' performance is largely dependent on its usage. On unmodified files queries were reported to evaluate "instantaneously." However, modifying files caused that "modifications stack up" causing high CPU usage and generally "very slow" responses. Besides, documentation was reported as slow to resolve while the server itself was "generally fast."

1.3.2 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. This section will first introduce the dataset before looking at the general performance and finally looking into particular cases.

1.3.2.1 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 distribution 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
total	16761	

Figures 1.2 break up these numbers by method and linearization size or file size respectively. The linearization is the linear representation of an enriched AST. It is explained in great detail in sec. **??**. 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 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

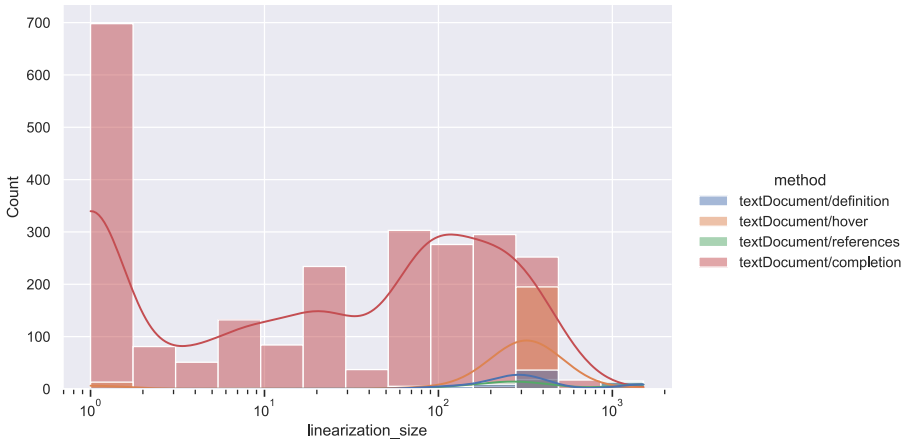


Figure 1.2: Distribution of linearization based LSP requests

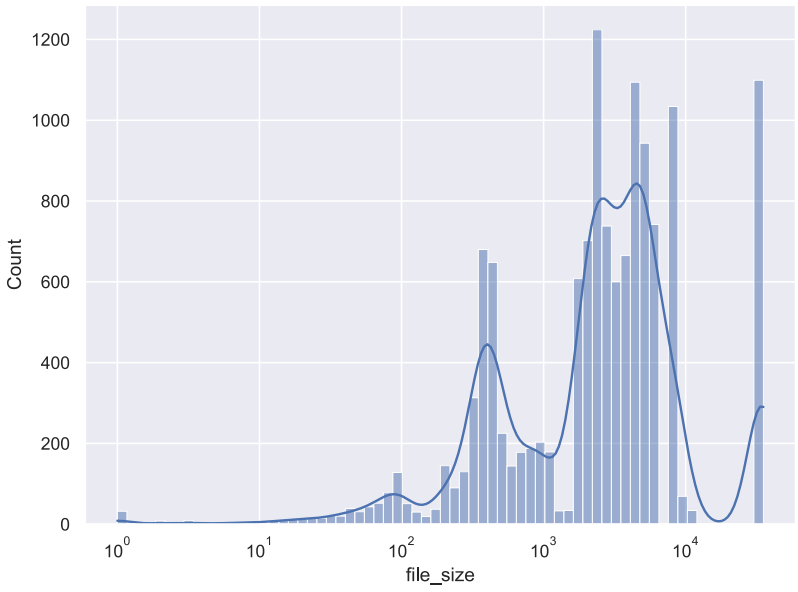


Figure 1.3: Distribution of file analysis requests

on 35 kilobyte large data.

1.3.2.2 Big Picture Latencies

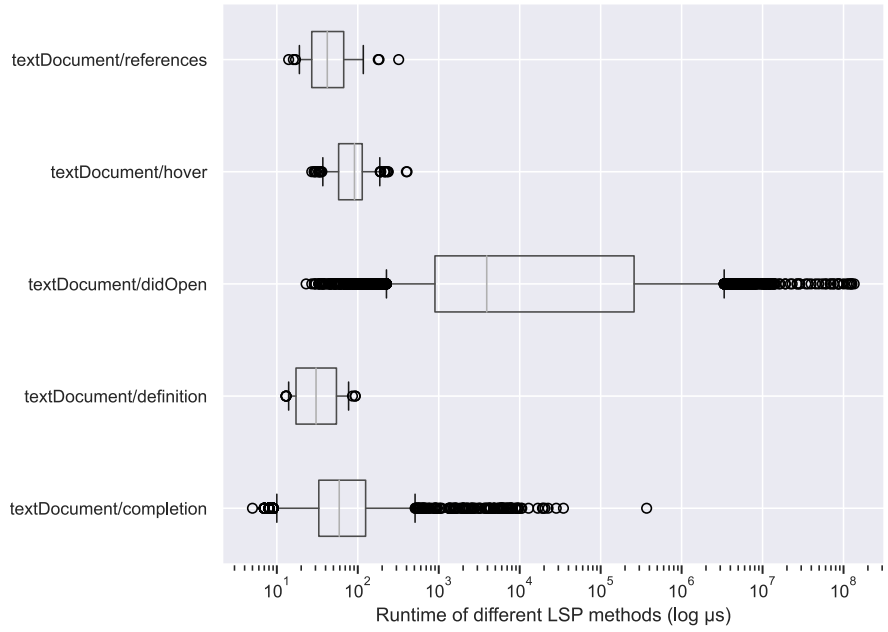
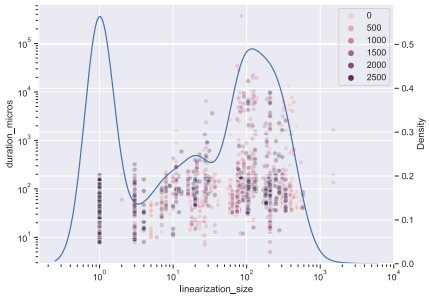


Figure 1.4: Statistical runtime of different LSP methods

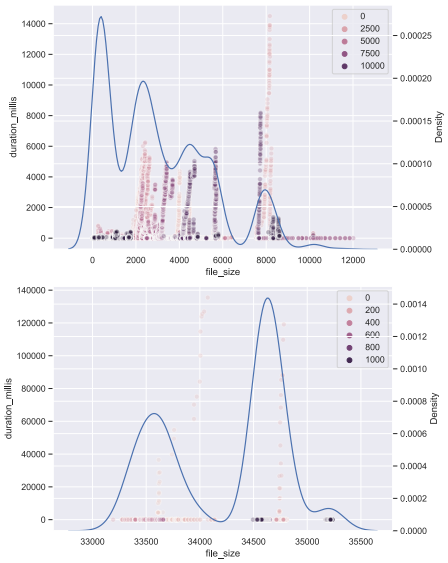
Comparing the runtime of the individual methods alone in fig. 1.4, reveals three key findings. First, all linearization based methods exhibit a sub-millisecond latency in at least 95 of all invocations and median response times of less than $100ţs$ However, maximum latencies of completion invocations 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 interquartile range spanning more than two orders of magnitude.

1.3.2.3 Special cases

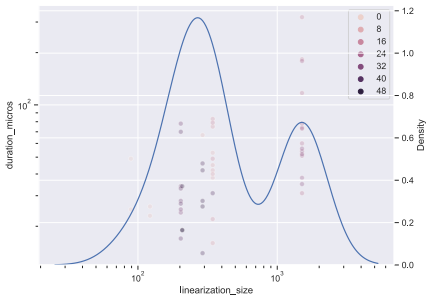
Setting the runtime of completion requests in relation to the linearization size on which the command was performed shows no clear correlation between the dimensions. In fact the correlation coefficient between both variables measures 0.01617 on a linear scale and 0.26 on a $\log_{10} \log_{10}$ scale. Instead, vertical columns stand out in the correlation graph fig. 1.5a. The height of these columns varies from one to five orders of magnitude. 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 reveals that the trace points of each column were recorded at a short time interval. Applying the same analysis to the other methods in figs. 1.5b, 1.5c, **??** returns similar findings, although the columns remain more compact in comparison to the Completions method. In case



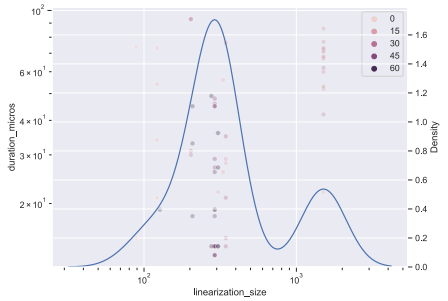
(a) Runtime latencies of completion requests at different linearization sizes



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



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



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

Figure 1.5: Runtime latencies of different linearization based methods

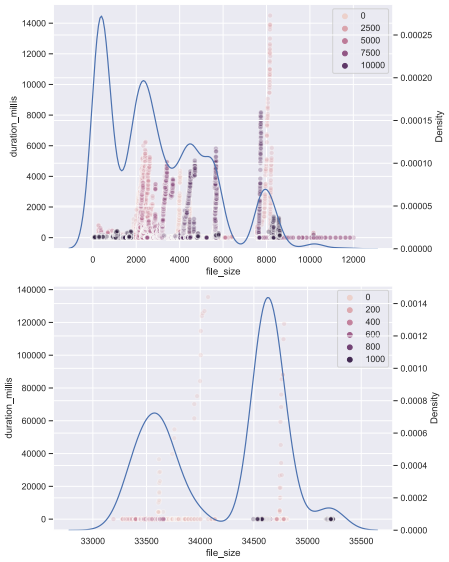


Figure 1.6: Runtime latencies of file update handling at different file sizes

of the `didOpen` method columns are clearly visible too [#fig:correlation-opens]. 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.

1.4 Discussion

This section discusses the issues raised during the survey and uncovered through the performance tracing. In the first part the individual findings are summarized and if possible grouped by their common cause. The second part addresses each cause and connects it to the relevant architecture decisions, while explaining the reason for it and discussing possible alternatives.

During the qualitative evaluation several features did not meet the expectations of the users. The survey also hinted performance issues that were solidified by the results of the quantitative analysis.

1.4.1 Diagnostics

First, participants criticized sec. 1.3.1.1.2 the diagnostics feature for some unhelpful error messages and specifically for not taking into account Nickel's hallmark feature, Contracts sec. **??**. While Contracts are a central element of Nickel and relied upon to validate data, the language server does not actually warn about contract breaches. Yet, while contracts and their application looks similar to types, contracts are a dynamic language element which are dynamically applied during evaluation. Therefore, it is not possible to determine whether a value conforms to a contract without evaluation of the contract. NLS's is integrated with Nickel's type-checking mechanism which precedes evaluation and provides only a static representation of the source code. In order to support diagnostics

for contracts NLS would need to locally evaluate arbitrary code that makes up contracts. However, contracts can not be evaluated entirely locally as they may transitively depend on other contracts. This is particularly true for a file's output value. Additionally, Contracts can implement any sort of complex computation including unbound recursion. Due to these caveats, evaluating contracts as part of NLS's analysis implies the evaluation of the entire code which was considered a possibly significant impact to the performance. As laid out above evaluating contracts locally is no option either. It is not only challenging to collect the minimal context of the Contract, the context may in fact be the entire program. An alternative option is to provide the ability to apply contracts manually using an LSP feature called "Code Lenses." Code Lenses are displayed by editors as annotations allowing the user to manually execute an associated action.

1.4.2 Cross File Navigation

In both cases `Jump-To-Definition` and `Find-References` surveyed users requested support for cross file navigation. In particular, finding the definition of a record field of an imported record should navigate the editor to the respective file as symbolized in lst. 1.1.

Listing 1.1 Minimal example of cross file referencing

```
// file_a.ncl

let b = import "./b.ncl" in b.field
                              |
                              +------+
                                     |
-----------------------------------  |
                                     |
// file_b.ncl                        |
                                     |
{                                    |
  field = "field value";             |
}  ^                                 |
   +---------------------------------+
```

The resolution of imported values is done at evaluation time, the AST therefore only contains nodes representing the concept of an import but no not reference elements of that file. NLS does ingest the AST without resolving these imports manually. The type checking module underlying NLS still recurses into imported files to check their formal correctness. As a result it would be possible for a NLS to resolve these links as an additional step in the post-processing by either inserting artificial linearization items sec. **??** or merging both files' linearization entirely.

1.4.3 Autocompletion

Another criticized element of NLS was the autocompletion feature. In the survey, participants mentioned the lack of additional information and distinction of

elements as well as NLS inability to provide completion for record fields. In Nickel, record access is declared by a period. An LSP client can to configured to ask for completions when such an access character is entered additionally to manual requests by the user. The language server is then responsible to provide a list of completion candidates depending on the context, i.e. the position. [Section #sec:completion] describes how NLS resolves this kind of request. NLS just lists all identifiers of declarations that are in scope at the given position. Notably, it does not take the preceding element into account as additional context. To support completing records, the server must first be aware of separating tokens such as the period symbol, check whether the current position is part of a token that is preceded by a separator and finally resolve the parent element to a record.

1.4.4 Performance

In the experience survey performance was pointed out as a potential issue. Especially in connection with the diagnostics and hover feature. NLS was described to "queue a lot of work and not respond" and show different performance signatures depending on its usage. While commands resolved "instantaneously" on unmodified files, editing a file causes high CPU usage and generally "very slow" responses. An analysis of the measured runtime of 16761 requests confirmed that observation. Both Hover and Update requests showed a wide range of latencies with some reaching more than two minutes. However, the data distribution also confirmed that latencies for most requests except `didOpen` are distributed well below one millisecond. The `didOpen` requests which are associated with the linearization process sec. **??** peak around $1ms$ but longer latencies remain frequent fig. **??**. Looking deeper into the individual features, reveals signs of the aforementioned "stacking." As discussed in sec. 1.3.2.3 subsequent requests exhibit increasing processing times especially during peak usage.

This behavior is caused by the architecture of the LSP and NLS' processing method. The Language Server Protocol is a synchronous protocol which requires the processing of all requests FIFO order. In effect, every request is delayed until previous requests are handled. This effect is particularly strong as the server is faced with a high volume. In the case of the trace for `didOpen` events the delay effect is greater than for other methods as `didOpen` is associated with a full analysis of the entire file. NLS architecture is heavily influenced by the desire to reuse as many elements of the Nickel runtime as possible to maintain feature parity with the evolving language core. Consequently, file updates invoke a complete eager analysis of the contents; The entire document is parsed, type checked and recorded to a linearization every time. In contrast, all other methods rely on the linearization of a document which allows them to use a binary search to efficiently lookup elements in logarithmic time. Additionally, all requests regardless of their type are subject to the same queue. Given that `didOpen` requests make up > 80 of the recorded events, suggests that other events are heavily slowed down collaterally.

Multiple ways exist to address this issue by reducing the average queue size. The most approachable way to reduce queue sizes is to reduce the number of requests the server needs to handle. The `didOpen` trace elements actually represents the joint processing path of initial file openings, and changes. NLS configures clients to signal changes both on save and following editor defined "change." The fact

that it is the editor's responsibility to define what constitutes a change means that some editors send invoke the server on every key press. In fig. 1.6 signs for such a behavior can be seen as local increases of processing time as the document grows. Hence, restricting analysis to happen only as the user saves the document could potentially reduce the load of requests substantially. Yet, many users preferred automatic processing to happen while they type. To server this pattern, NLS could implement a debouncing mechanism for the processing of document changes. The messages associated to document changes and openings are technically no requests but notifications. The specification of JSON-RPC which the LSP is based on defines that notifications are not allowing a server response. Clients can not rely on the execution of associated procedures. In effect, a language server like NLS, where each change notification contains the entire latest document, may skip the processing of changes. In practice, NLS could skip such queue items if a more recent version of the file is notified later in the queue. The queue size can also be influenced by reducing the processing time. Other language servers such as the rust-analyzer (**rust-analyzer?**) chose to process documents lazily. Update requests incrementally change an internal model which other requests use as a basis to invoke targeted analysis, resolve elements and more. The entire model is based on in incremental computation model which automates memorization of requests. This method however requires rust-analyzer to reimplement core components of rust to support incrementally. Therefore, if one accepts to implement an incremental model of Nickel (including parsing and type checking) to enable incremental analysis in NLS, switching to a lazy model is a viable method to reduce the processing time of change notifications and shorten the queue.