UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Taaniel Saarnik

Leveraging the First Futamura Projection for Large-scale Rule Parallelisation in an Industrial Datalog Engine

Master’s Thesis (30 ECTS)

Supervisors:	Bruno Rucy Carneiro Alves de Lima, MSc
	Mykhailo Dorokhov, MSc

Tartu 2024

Leveraging the First Futamura Projection for Large-scale Rule Parallelisation in an Industrial Datalog Engine

Abstract:
Incorrect system configurations can cause disruptions in the software development lifecycle from code deployment to system reliability. The correctness of these configurations can be ensured by an on-site DevOps team or by the developers themselves. A system that finds the incorrect settings automatically could be highly beneficial.

A company's internal system called Neodora uses the Open Policy Agent top-down Datalog engine to check the correctness of repositories and code in pull requests. It gives developers a faster and easier way to discover and correct mistakes. While the current implementation fulfils its purpose, it struggles with the near real-time scale that it is subjected to, leading to high costs.

We present an industrial experience report detailing how a new implementation of Neodora eliminates this weak point by leveraging the first Futamura projection to cleverly attain large-scale rule parallelisation. Our changes to Neodora led to a potential reduction in cost of nearly fifteen times.

Keywords:
Datalog, Open Policy Agent, Rego, Futamura projection, partial evaluation

CERCS: P170 - Computer science, numerical analysis, systems, control

Tööstuslikus andmelogimootoris Futamura esimese projektsiooni kasutamine suuremahulise reeglite paralleelsuse saavutamiseks

Lühikokkuvõte: Süsteemi valed konfiguratsioonid võivad põhjustada palju probleeme alates käivitamisest kuni süsteemi töökindluseni. Nende seadistuste õigsuse eest võib vastutada DevOpsi meeskond, aga seda vastutust võib laiendada ka arendajatele. Viimane jaotab töökoormuse paljude inimeste vahel ja õigesti tehes tõstab see produktiivsust tervikuna. Arendajate abistamiseks selle ülesande täitmisel on väga kasu süsteemist, mis leiab valed sätted automaatselt. Üks selline ettevõtte sisemine tööriist kannab nime Neodora, mis kasutab Open Policy Agent-it koodihoidlate (ing *repository*) ja tõmbekutsete (ing *pull request*) koodi õigsuse kontrollimiseks. See annab arendajatele kiirema ja lihtsama viisi vigade avastamiseks ja parandamiseks.

Kuigi Neodora täidab oma eesmärki edukalt, selle jõudlust saaks ikkagi suurendada. Neodora töötab suurepäraselt üksikute koodihoidlate ja tõmbetaotluste puhul, kuid raskusi tekitab Bulk-Neodora teostus. Bulk-Neodora on mõeldud igapäevaseks ettevõtte koodihoidlate skaneerimiseks. Seda tehakse käivitades Neodorat iga koodihoidla peal. Mõned Neodora reeglid teevad aga HTTP-päringuid, mis on samad olenemata sellest, millist koodihoidlat skaneeritakse.

Esitleme Neodora uut versiooni, mis kõrvaldab selle nõrga koha, vähendades Bulk-Neodora tööaega märkimisväärselt. See saavutati Futamura esimese projektsiooni ja paralleelsuse kasutamisega. Lisaks asendati kettatoimingud (ing *disc operations*) kiiremate mälusiseste (ing *in-memory*) toimingutega. Muudatused Neodoras vähendasid Bulk-Neodora tööaega peaaegu viisteist korda.

Võtmesõnad:
Datalog, Open Policy Agent, Rego, Futamura projektsioon, osaline hindamine

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1 Introduction

Configuration information is used by systems and services to decide how to operate. It includes, but is not limited to, starting up, shutting down and interacting with other systems or services. The configuration may be suitable out of the box, but sometimes custom configuration is needed. However, changing configurations can lead to various problems if not done correctly.

Bad configuration can lead to the system not starting, running poorly, having unexpected downtime or data loss. The chance of mishaps rises when it is required to configure multiple systems or services, for example, in systems that consist of several microservices. Every microservice can have some configurations that are the same as for others but also service-specific. Depending on the company, different teams take care of their services, but it can be limited to the service implementation and logic. The developers may not handle the configuration, specifying how it should be run together with other services. For this, a DevOps team is responsible for managing services' configurations.

DevOps team can be a suitable solution when the service count and uniqueness are small; otherwise, it can be overwhelming. Every month, thousands of repositories get hundreds of thousands of changes, and doing it manually is time-consuming and error-prone. This created a need for a system to validate service configurations in real time and quickly notify developers if something should be changed. It was solved by developing a system called Neodora [1].

The core of Neodora is Open Policy Agent (OPA), a high-performance Datalog engine with JSON semantics. Datalog is a declarative programming language where rules describing relationships between data elements can be defined. While Neodora itself fulfils its purpose, periodically running it over all repositories in a company (called Bulk-Neodora execution) to check if all the rules are satisfied can take a considerable amount of time.

This paper provides some background on Datalog, Open Policy Agent, Rego, and Futamura projections. We also look into previous works that use partial evaluation. In addition, we describe the previous solution and Neodora, bringing out their strong and weak points. Finally, we look into what takes the most time for Neodora's execution.

This work aimed to propose a solution using the first Futamura projection and parallelisation to speed up the periodical Bulk-Neodora execution that validates all rules over all the repositories. The research questions to answer were:

1. **Where is the most time spent during a Neodora or Open Policy Agent rule execution?**

2. **Could some work be done utilizing partial evaluation, shifting repeated code and computation from runtime to compile time?**

3. **Are there any operations that could be improved to decrease the program's runtime?**

To find answers to these questions, Neodora's execution time was inspected, and parts of the code that could be improved to save time were explored. The main contribution of this thesis lies in how large-scale Datalog systems that work on high amounts of data and rely on fetched dynamic information could be improved using partial evaluation and parallelisation. More specifically, we are looking into how Open Policy Agent and its rules could be written to support it. However, the same idea could be used for other cases.

2 Background

This section focuses on three main components that are relevant to Neodora. It also describes the first Futamura projection and how it can help improve Bulk-Neodora execution. In addition, to help understand the code examples in this paper, a short introduction to the Go programming language is given.

2.1 Datalog

Datalog is a declarative language that uses horn clauses to evaluate facts. It allows one to make queries with sentences containing variables, defining conditions and using those to evaluate a rule or decision. Using either top-down or bottom-up evaluation for Datalog [2] is possible. To demonstrate their difference, let's consider the following rules and facts (1) for the transitive closure of a simple graph [2]. Rules in Datalog are read from right to left, meaning if the right side is true, then the left side is also true. Commas can be read as logical *AND* operators.

$$\begin{aligned} & \leftarrow \text{table} \quad \text{p/2} \\ \text{Rules:}\ & \mathbf{p}(X,Y) \leftarrow \mathbf{e}(X,Y). \\ & \mathbf{p}(X,Y) \leftarrow \mathbf{p}(X,Z), \mathbf{e}(Z,Y). \\ \text{Facts:}\ & \mathbf{e}(a,b). \quad \mathbf{e}(e,a). \\ & \mathbf{e}(d,e). \quad \mathbf{e}(b,c). \quad \mathbf{e}(c,b). \end{aligned} \tag{1}$$

Under facts, we have edges that are known. The first rule says that every edge (**e**) is a path (**p**), and the second rule creates new paths from edges that are already known or discovered during the evaluation. The top-down evaluation starts with the query and works its way down to find the answers. The top-down evaluation can be [2]:

$$\begin{array}{lll} 1. & \mathbf{p}(a,A) & \text{add query } \mathbf{p}(a,?) \text{ to table} \\ 2. & \quad \mathbf{e}(a,A) & \text{resolve 1 with 1st rule} \\ 3. & \qquad ()A=b & \text{resolve 2 with fact } \mathbf{e}(a,b)\text{, add answer } \mathbf{p}(a,b) \text{ to table} \\ 4. & \quad \mathbf{p}(a,Z),\mathbf{e}(Z,A) & \text{resolve 1 with 2nd rule} \\ 5. & \qquad \mathbf{e}(b,A) & \text{resolve 4 with answer } \mathbf{p}(a,b) \text{ from table} \\ 6. & \qquad\quad ()A=c & \text{resolve 5 with fact } \mathbf{e}(b,c)\text{, add answer } \mathbf{p}(a,c) \text{ to table} \\ 7. & \qquad \mathbf{e}(c,A) & \text{resolve 4 with answer } \mathbf{p}(a,c) \text{ from table} \\ 8. & \qquad\quad ()A=b & \text{resolve 7 with fact } \mathbf{e}(c,b)\text{, } \mathbf{p}(a,b) \text{ in table, don't add} \end{array} \tag{2}$$

The query $\mathbf{p}(a, A)$ is used to evaluate rules. New facts are created using the first rule and then using the second rule. The top-down evaluation finds only the facts that are relevant to the query. The same query in bottom-up evaluation can be [2]:

Iteration 0: infer the program facts:

$\mathbf{e}(a,b), \mathbf{e}(e,a), \mathbf{e}(d,e), \mathbf{e}(b,c), \mathbf{e}(c,b)$

Iteration 1: Use those facts and program rule 1 to infer:

new: $\mathbf{p}(a,b), \mathbf{p}(e,a), \mathbf{p}(d,e), \mathbf{p}(b,c), \mathbf{p}(c,b)$

Iteration 2: Use rule 2 and previous inferred facts to infer:

new: $\mathbf{p}(a,c)$ [from $\mathbf{p}(a,b)$ and $\mathbf{e}(b,c)$]

$\mathbf{p}(e,b)$ [from $\mathbf{p}(e,a)$ and $\mathbf{e}(a,b)$]

$\mathbf{p}(d,a)$ [from $\mathbf{p}(d,e)$ and $\mathbf{e}(e,a)$]

$\mathbf{p}(b,b)$ [from $\mathbf{p}(b,c)$ and $\mathbf{e}(c,b)$]

$\mathbf{p}(c,c)$ [from $\mathbf{p}(c,b)$ and $\mathbf{e}(b,c)$] (3)

Iteration 3: Again use rule 2 with previously inferred facts to infer:

new: $\mathbf{p}(e,c)$ [from $\mathbf{p}(e,b)$ and $\mathbf{e}(b,c)$]

$\mathbf{p}(d,b)$ [from $\mathbf{p}(d,a)$ and $\mathbf{e}(a,b)$]

(We get duplicates, e.g., $\mathbf{p}(a,b)$ from $\mathbf{p}(a,c)$ and $\mathbf{e}(c,b)$, but they aren't added.)

Iteration 4: Again use rule 2

$\mathbf{p}(d,c)$ [from $\mathbf{p}(d,b)$ and $\mathbf{e}(b,c)$]

Iteration 5: Nothing new to infer, so stop

In bottom-up evaluation, all the possible paths are found. The query can be answered by looking up the paths with a as the first value.

Comparing these two evaluation techniques, we can see that the bottom-up evaluation also finds paths that are not required to answer the query [2]. The top-down and bottom-up evaluations can answer the same queries but are mainly used for different goals. The fact that bottom-up evaluation finds all facts becomes useful when multiple queries with different arguments are done. This way, the facts can be gathered once, and the answers to the queries can be found quickly. In addition, while top-down implementation answers if the fact provided by the user is true or false, the bottom-up implementation can help find new facts.

Neodora, introduced later in this paper, uses top-down evaluation for Datalog. A simple example of a rule that it could check is shown next. Let's say we want to define a rule (4) that checks one package's dependence on another; we can define a rule **requires** that checks if package *X* requires a package *Y*:

$$\textbf{depends}(X, Y) \leftarrow \textbf{requires}(X, Y) \tag{4}$$

By using top-down evaluation, we can get the output to be true or false depending on what the rule evaluates to. The rule in example 1 is evaluated as true if the query is $? \leftarrow$ **depends**(*X*, *Y*) and a fact **requires**(*X*, *Y*) is true. The previous rule can be extended to evaluate the dependency of connected packages by adding another rule (5):

$$\textbf{depends}(X, Y) \leftarrow \textbf{depends}(X, Z), \textbf{depends}(Z, Y) \tag{5}$$

By adding another rule, we can make use of recursion, and upon evaluating the first rule, we have a fact for the second rule. If there are two facts **requires**("Package1", "Package2") and **requires**("Package2", "Package3"), then when we query **depends**("Package1", "Package3"), a new connection can be made between "Package1" and "Package3" without explicitly defining it.

2.2 Open Policy Agent

Open Policy Agent (OPA) [3] is a general-purpose policy engine providing high-level declarative language to specify policy as a code. It can validate data using defined policies (rules) and additional data, which can, for example, be accepted values for some variables. OPA is not restricted to a certain domain, allowing it to be used to enforce many different policies. When the facts are evaluated with respect to the program, the results are returned. The results can be simple boolean values or arbitrary structured data. In addition to that, it supports features you can expect from a programming language, such as testing and coverage. The support for policy testing makes creating and modifying policies easier. The testing format is similar to typical programming languages. Policies (example in listing 5) and tests (example in listing 6) are contained in separate files, each test prefixed with *test_* so that they would be discovered when tests are run.

2.3 Go

To help better understand the code examples in this paper, this section briefly introduces Go, also known as Golang [4]. Go was developed by Google in 2007. It has a similar syntax to C and OPA's policy language, Rego. It is easy to read, has static types, and supports garbage collection and concurrency. In addition, it already has many packages that can be used to complete various tasks. The following listing 1 is used to explain the Go language:

```
 1 package main
 2
 3 import "fmt"
 4
 5 type Person struct {
 6   name     string
 7   location string
 8   age      int
 9   info     string
10 }
11
12 func getGreeting(person Person, tellMore bool) string {
13   if person.info != "" && tellMore {
14     return fmt.Sprintf("Hello, I'm %s from %s. %s", person.name
     , person.location, person.info)
15   } else {
16     return fmt.Sprintf("Hello, I'm %s from %s.", person.name,
    person.location)
17   }
18 }
19
20 func main() {
21   const name = "Tom"
22   fmt.Printf("Hello %s!\n", name)
23   name2 := "Peter"
24   fmt.Printf("Hello %s!\n", name2)
25   name2 = "Alice"
26   fmt.Printf("Hello %s!\n", name2)
27
28   var person Person
29   person = Person{name: name, location: "Earth", age: 32, info:
      "I like Go."}
30   greeting := getGreeting(person, false)
31   fmt.Println(greeting)
32   greeting = getGreeting(person, true)
33   fmt.Println(greeting)
34 }
```

Listing 1. Go example.

The code example shows different ways to define variables and types and use packages and functions. On line 1, a package is defined, and on line 3, another package is imported. Packages are used to organize code, allowing one to import code from one's own project or the library (third-party packages). This example uses a package *fmt* to create and print sentences. In addition to default variable types (string, bool, int, etc.), it is possible to create structs (lines 5-10) that can be used similarly to types, as done on line 28. Type *Person* has a name, location, info value as a string, and age as an integer. The types of variables can be explicitly specified (line 28) or left for a compiler to infer them (for example, lines 21, 23, 30). Keyword *var* can also be skipped when a variable is initialized with a value using the *:=* operator. Variables with *const* keyword can not have their values reassigned. A function *getGreeting* (lines 12-18) takes in two variables, a *Person* and a boolean, and returns a string. The function returns an extended greeting text if the *Person* has info and (&& - logical AND operator) the boolean is *true*. Otherwise, only name and location are mentioned in the greeting. Running the code outputs the following:

```
Hello Tom!
Hello Peter!
Hello Alice!
Hello, I'm Tom from Earth.
Hello, I'm Tom from Earth. I like Go.
```

Listing 2. Go example output.

The first three sentences are printed using only the name variable. The last two sentences use the *getGreeting* function to create a desired greeting.

2.4 Rego

Open Policy Agent uses its declarative language Rego [3], designed to express policies over data structures. It supports a wide range of data-handling operations.

```
faultyImages := {"Faulty-dev", "Image-dev-1.3"}
deny contains msg if {
    some service in input.services
    service.image in faultyImages
    msg := sprintf("'%v' is not allowed in a service '%v'.", [
   service.image, service.name])
}
```

Listing 3. Rego example.

Listing 3 shows a rule that checks if the services' images are considered faulty. In the first line, we have defined two images that should not be used. From the second line, we have a rule that checks services' images given as input. The input is a JSON document containing a key *"services"* with a list of services containing their image as a value. Keyword *some* is used to declare a local variable explicitly. In the example, we loop over services and assign a new value using *some*. Lastly, we check if the service's image is in the faulty images list. If true for a service, the message gets added to the result.

```
{
    "deny": [
        "'dev-1.3' is not allowed in a service 'Service2'."
    ],
    "faultyImages": [
        "dev",
        "dev-1.3"
    ]
}
```

Listing 4. Rego example output.

Listing 4 is an example output. In that case, *Service2* had an incorrect image. The deny value will be empty if all services use an allowed image. The same rule in Datalog (6) could be:

$$\textbf{hasFaultyImage}(X) \leftarrow \textbf{services}(X), \textbf{isIn}(X, faultyImages) \tag{6}$$

Defining rules and reading them is simplified in Rego. It also supports structured document models such as JSON, providing simple ways to traverse them.

A new file can be created to test a policy. A good option would be to keep all the single policy files in a separate folder named after the policy to have a clear file structure. The policy test file can contain multiple tests, each prefixed with *test_*, to cover all the possible cases related to the policy. The following listings 5 and 6 show how testing can be done in Rego.

```
allow if {
  some x in data.policies
  x.name == "test_policy"
  matches_role(input.role)
}
matches_role(my_role) if input.user in data.roles[my_role]
```

Listing 5. Policy example [5].

The policy checks if the user's role that is given as input matches the defined user role. Information in *data* is a part of policy definition, data that it uses, and *input* is what is given to make a decision. The *data* can be the file's contents that are read in and used when policy is being evaluated.

```
policies := [{"name": "test_policy"}]
roles := {"admin": ["alice"]}
```

```
test_allow_with_data if {
  allow with input as {"user": "alice", "role": "admin"}
    with data.policies as policies
    with data.roles as roles
}
```

Listing 6. Policy test example [5].

The test in listing 6 is about the previously shown policy. OPA supports data and function mocking. In this test, a keyword *with* is used to mock *data* and *input* values, allowing the evaluation of the policy with different values. The result of a test is either PASS, FAIL, ERROR (runtime error) or SKIPPED (for tests prefixed with *todo_*).

```
$ opa test pass_fail_error_test.rego
data.example.test_failure: FAIL (253ns)
data.example.test_error: ERROR (289ns)
  pass_fail_error_test.rego:15: eval_builtin_error: div: divide
    by zero
--------------------------------------------------------------
PASS: 1/3
FAIL: 1/3
ERROR: 1/3
```

Listing 7. Executing tests in a command-line interface [5].

Listing 7 shows a command-line output when the tests are run. The output format can also be changed to JSON, which is useful when integrating OPA with other systems.

2.5 First Futamura Projection

Futamura projections are types of partial evaluations that can be used for program optimization [6]. In total, there are three Futamura projections. However, only the first is relevant to this current paper. The first Futamura projection was devised by Yoshihiko Futamura in 1971, and it states that specialising an interpreter for a given source code yields an executable. Given a program, we can create an interpreter that takes in input and returns a specialised program that takes in the remaining input, which returns output the same as the original program. Because some of the work is already done by specialising an interpreter, the specialised program is expected to run faster than the original.

Let *p* be a program and *i1* and *i2* inputs to this program [7]. We can define the performance increase like so:

$$\textbf{time}_{p_{i1}}(i2) < \textbf{time}_{p}(i1, i2) \tag{7}$$

The $\textbf{time}_x(y)$ is a numerical value meaning runtime of the program on a given input, and p_{i1} is a program specialised on input *i1*. This runtime performance increase comes with the cost of increased compilation time. However, if the compiled program is used multiple times with different inputs, there is a great payoff. In our case, Neodora builds and executes the Open Policy Agent. The basic configuration of Neodora and OPA stays the same between different repository scans, and many rules need the same data, such as some URL request responses. Let's call this data *iStatic*. This means the interpreter can be specialised to the input *iStatic*, giving a program *prog** [8]. *prog** is a compiled version of *iStatic* so it does not need to be supplied again. Instead, we can only provide the remaining data for computing.

We can analyze our program to see what operations are always done over all the repositories. Instead of doing these for every repository, we can do it only once and have it in a compiled version.

3 Related works

The idea of program specialisation using Futamura projections is not new. Multiple works have been done on different programming languages and also on Datalog. This section of the paper looks more into partial evaluation and a few recent works related to this paper.

3.1 Partial evaluation

A book by Neil D. Jones and others [9] talks about partial evaluation, highlighting general principles and giving examples of its use. It delves into partial evaluation, explaining its ties to program specialisation. Suppose we have a subject program, and that program's input has static data for any input. Then, we can have a partial evaluator, shown by the following equations [9], that creates a specialised program with all the operations on that static data already done. In the equation 8, *p* stands for the subject program, and a partial evaluator is called *mix*.

$$\begin{aligned} \text{Computation in one stage: } output &= [p][input1, input2] \\ \text{in two stages: } p_{input1} &= [mix][p, input1] \\ output &= [p_{input1}]input2 \\ \text{Definition of partial evaluation: } [p][input1, input2] &= [\underbrace{[mix][p, input1]}_{\text{specialised program}}]input2 \end{aligned} \tag{8}$$

To execute the program, only the remaining dynamic data must be given to the specialised program to get the output. This way, we already have data from operations done on static data in the compiled program. This means we can expect a speedup when executing the program because it has less work to do. If the output is the same as for the original program, then the specialised program works as intended. The main idea of partial evaluation can be shown using code. Listing 8 shows an example of a code that does some calculations on two numbers and returns them. It also requests a secret number (let that be 4) and uses it to modify return values.

```
func fetchSecretNumber() (int) {
    // Long request to get a daily secret number.
    return 4
}

func calc( x int,  y int) (int, int) {
    secretNumber := fetchSecretNumber()
    add := x + y
    sub := x - y
    return add + secretNumber, sub + secretNumber
}

func main() {
    fmt.Println(calc(10, 2)) // 16 12
    fmt.Println(calc(8, 4))  // 16 8
}
```

Listing 8. Original code.

Every day, a new secret number can be accessed from the Internet. If we were to call this function multiple times, a lot of time is spent just waiting for data. Listing 9 displays code where partial evaluation is used.

```
func fetchSecretNumber() (int) {
    // Long request to get a daily secret number.
    return 4
}

func calc() func(x int, y int) (int, int) {
    secretNumber := fetchSecretNumber()
    return func(x int, y int) (int, int){
      add := x + y
      sub := x - y
      return add + secretNumber, sub + secretNumber
    }
}

func main() {
    calcWithSecret := calc()
    fmt.Println(calcWithSecret(10, 2)) // 16 12
    fmt.Println(calcWithSecret(8, 4))  // 16 8
}
```

Listing 9. Updated code with partial evaluation.

This way, the secret number is requested only once, and calculations can be made many times without waiting for the request. The same logic can be used on a whole program. We can reduce the number of operations by calculating static values and/or fetching data in advance, assuming they are not expected to change frequently. We can run the program with different dynamic inputs, and the impact of this becomes clearer the more we run it.

In addition, they [9] bring out that although partial evaluation has some promising applications and works well in practice, there are still some possible problems. The performance increase depends greatly on the base program and how the interpreter is written. Dynamic name binding and dynamic code creation cause the program to do runtime variable name searches or contain runtime source language text. It is also hard to predict the speedup by specialising the program and ensuring that the specialised program works as expected based on a given input.

3.2 Parallel evaluation of C++ constant expressions

Evaluating many expressions during compile time can increase runtime performance, but it comes with increased compile time. A conference paper by A. Gozillon and others [10] looks into this problem and proposes a method to levitate some of this in the programming language C++. They bring out that C++ has *constexpr* specifier, allowing functions and variable declarations to be evaluated at compile time. They introduced ClangOz, a compiler that supports the parallel execution of *std::for_each* loops at compile time. They ran five compile-time benchmarks to test their compiler. Although the average performance benefit was above 50% and reached 100% in one of the benchmarks, there are still a few problems and limitations to address. Most notable is that it does not support nested loop parallelism, and only one loop in every function can be parallelised. In addition, cloning and distributing data among threads can be expensive, increasing startup costs. When these issues are addressed, the performance gains could be higher.

3.3 Soufflé

A paper published by H. Jordan and others [11] introduces a tool called Soufflé to overcome the performance limitations of Datalog evaluation. More precisely, they saw that multiple tools for program analysis that had been developed previously were created with a "one size fits all" mindset. While it allows the use of these tools for every possible use case, the tools lack the ability to specialise their evaluation process for a certain program analysis specification, leaving out possible performance gains.

Soufflé is currently being used at Oracle Labs for Java security analyses because of its performance, ease of use and customizability [11]. They did it by synthesising Datalog specifications to a C++ program. To achieve this, three specialisations were made using the first Futamura projection. Firstly, they specialised Datalog itself, receiving a relational algebra machine that was further specialised to receive a C++ code with OpenMP for parallel execution support. The second specialisation's main goal was to improve the worst-case runtime complexity of the relational algebra machine. More precisely, they improved loop-based join operations and the identification of optimal index. Dilworth's theorem-inspired algorithm improved index management by computing only the necessary indices. The final specialisation was made on the C++ program received from the second step. Using efficient parallel versions of B-trees and Tries, they specialised data structures and algorithms by static information, increasing the runtime performance.

They [11] ran three analyses on the OpenJDK7 library, comparing Soufflé and three other Datalog evaluation tools (results shown in table 1).

Table 1. Soufflé test results [11]. Time is in hh:mm:ss and memory in GB.

	CI		CS		Security	
Tools	Time	Memory	Time	Memory	Time	Memory
bddbddb	0:30:00	5.7	DNF	DNF	DNF	DNF
SQLite	6:20:00	40.2	DNF	DNF	DNF	DNF
μZ	DNF	DNF	DNF	DNF	DNF	DNF
Soufflé	0:00:35	8.5	6:44:08	206.4	14:45:01	75.3

Soufflé greatly outperformed the other tools. It used fewer or similar resources and managed to analyse large code bases that others could not in a reasonable time frame. They set a reasonable time frame of 18 hours due to the availability of computation resources.

4 The setting

The necessity for a tool that can scan all the repositories and evaluate their acceptance against predefined rules comes from managing cloud-based software as a service containing hundreds of microservices. The company has hundreds of software engineers who are developing different parts of the software. It is important to have a set of rules automatically checked with every code update since engineers come and go, and nobody can remember all of them themselves. Using that kind of tool cuts down on development time and improves the security and performance of the software.

The company repositories are located in GitHub, along with a continuous integration system. Engineers work on microservices pulled from Github, make changes locally and then push a new branch to Github. When the code is ready to be published, a pull request is made to the master branch. The code to be merged with the master branch has to be reviewed by another engineer. Automated policy enforcement must take place to reduce unnecessary work for them.

4.1 Dora

For a while, a tool called Deployment or Repository Analyzer (Dora) [12] was used. It was created to increase Docker image and container security by analyzing Docker and Docker Compose files and validating them against Docker's best practices and defined policies.

```
module.exports = async function (config) {
  try {
    // Negative Lookahead on comments line
    const key = `^(?!#)${config.param}`;
    // Get Base image from Dockerfile.CI
    const dockerfileCIResults = await findStringByKey(config,
   key);
    const dockerFileConfig = { path: 'Dockerfile', param:
   config.param };
    const dockerfileResults = await findStringByKey(
   dockerFileConfig, key);

    if (dockerfileCIResults.length > 0) {
      // Compare the last FROM occurency for both files
      const baseImageCI = dockerfileCIResults.slice(-1)[0].line
   .value;
      if (dockerfileResults.length > 0) {
        // Image found in both files
        const baseDevImage = dockerfileResults.slice(-1)[0].
   line.value.trim();
16        if (baseImageCI !== baseDevImage && `${baseImageCI}-dev` !== baseDevImage && !isGoService()) {
17          await reporter.report(config, dockerfileCIResults.slice(-1)[0].line);
18        }
19      } else {
20        // Image found only in Dockerfile.CI
21        await reporter.report(config, dockerfileCIResults.slice(-1)[0].line);
22      }
23    } else {
24      if (dockerfileResults.length > 0) {
25        // Image found only in Dockerfile
26        await reporter.report(config, dockerfileResults.slice(-1)[0].line);
27      }
28    }
29  } catch (err) {
30    const error = `Error message: ${err}`;
31    await reporter.reportError(error, config);
32  }
33 };
```

Listing 10. Dora rule example [1].

Listing 10 shows a Dora rule that was used to compare parent images in *Dockerfile* and *Dockerfile.CI*. At first, on lines 4 to 8, the files are found and loaded in. The rule has three primary messages (lines 17, 21, 26) to report, depending on whether the image is found in both files or one of those. If something goes wrong, then an error is caught and reported. The necessity for a new tool came from having an excessive amount of boilerplate code in rules. The rules contained a lot of code that was not about rule validation but about setting up data for it. For example, the data gathering on the first lines and obtaining the values from them using slicing, indexes, and trimming (example on line 15). In addition, there was a need for other policies that were not about Docker.

4.2 Neodora

The new tool that replaced Dora is called Neodora [1]. It uses OPA, which reduces the complexity of writing and reading rules. In addition, both OPA and Neodora are written in Go. Listing 11 shows how the previously described rule in listing 10 is written in Neodora.

```
imageDefinition[result] {
  data.neodora.files["go.mod"] == null
  ciImage := parent_image(input)
  normalImage := trim_suffix(parent_image(data.neodora.files.
   Dockerfile), "-dev")
  ciImage != normalImage
  result = {
    "ciImage": ciImage,
    "normalImage": normalImage,
  }
}
```

Listing 11. Neodora example [1].

All the unnecessary boilerplate code is gone. Data required for this rule is accessed using *data.neodora.files*, which has the data already parsed based on the data type (JSON, text, YAML, Docker file etc.) and supports the required operations on them. This means the necessary data finding and reading are done in the background; the rules only have the rule logic. It helps keep the code clean and readable. Testing the rules was also made easier since OPA supports writing tests. Dora required to have a repository for failures and another repository for passes. However, with Neodora, it is possible to write tests in a separate file where you define an input to your rule as shown in listing 12.

```
test_allow_internal if {
  allow_internal with input as {"ip": "192.168.1.254"}
}
test_deny_external if {
  not allow_internal with input as {"ip": "178.128.73.145"}
}
```

Listing 12. Simple rule test example.

If the IP is private, the rule and the test pass. And if the IP is public, the rule fails, meaning the test has passed. Both input and additional data can be changed to write tests for the rules. Rule test files can be in the same directory as the rule file, keeping the file structure clear and code easy to modify.

Figure 1 shows the architecture of Neodora. It requires two compilations, one for Neodora's CLI and one for OPA. When CLI is compiled, the rules and their specifications

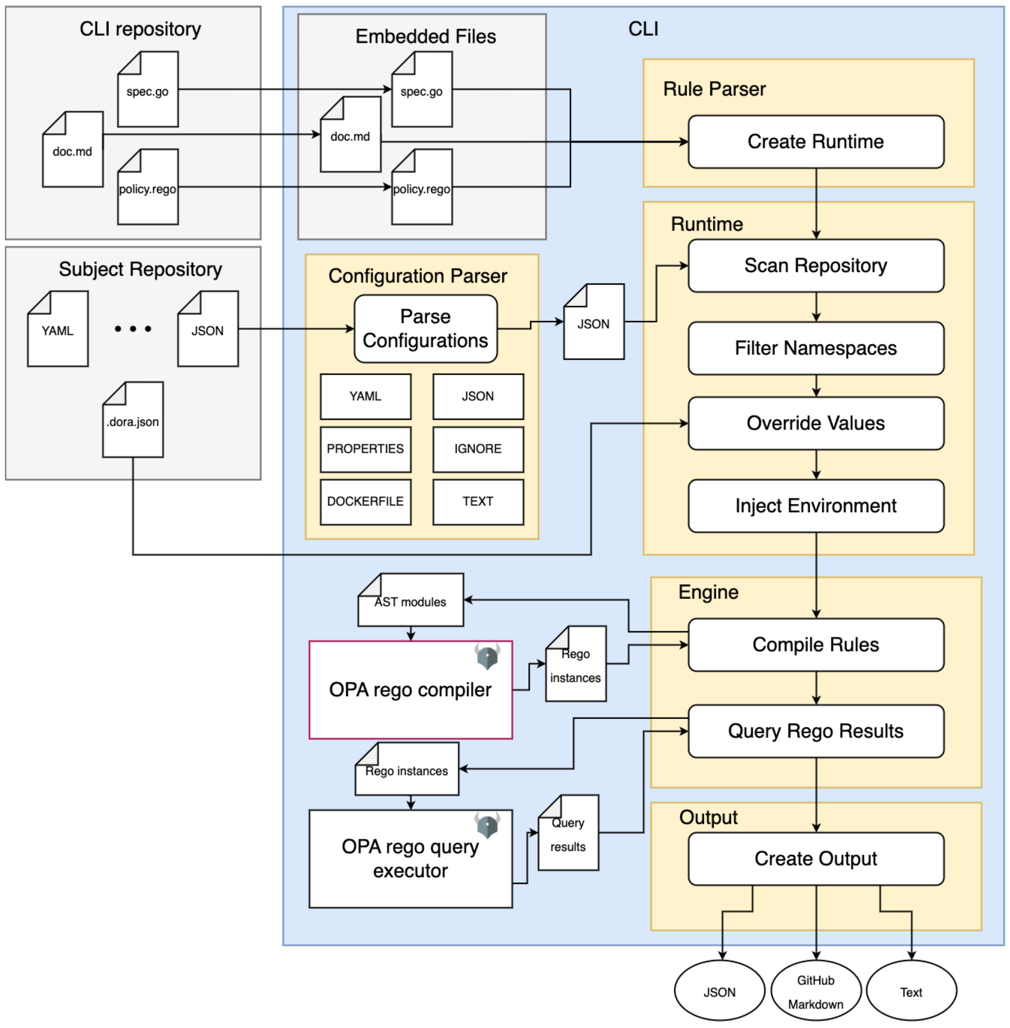

Figure 1. Neodora architecture [1].

are added as strings. The specification contains information about the rule. For example, it has rule ID, name, documentation, output format, active status, and input data and can include additional variables. When Neodora is run, OPA compilation takes place. The rules and specifications are injected into OPA. Since the repositories used in the company can differ highly, there was a need to adjust Neodora execution per repository. It was solved by allowing engineers to define a file (example in listing 13) that is read in to modify Neodora rule configurations.

```json
{
    "helmCpuReservation": {
        "upperLimitPct": 500
    },
    "sonarConfiguration": {
        "active": false
    },
    "helmCpuReservation": {
        "skipContainers": "messenger-redis"
    }
}
```

Listing 13. Repository Neodora config.

Not all the rules defined in Neodora are suitable for all repositories. Services can use different frameworks and have special requirements that could cause the Neodora repository check to fail and block the deployment. For example, a rule can be disabled if it is made clear that it is not relevant for a service or to bypass the check while the fix is being worked on. The configuration file allows us to eliminate these issues by describing rule configurations per repository.

The current Neodora implementation takes in a repository branch and runs the rules against it. It has been a game changer in the company. The ease of use, addition of new rules, and output have been sufficient. The only current weak point of Neodora has been running it over all the repositories in a certain interval, also known as Bulk-Neodora. Bulk-Neodora is a workflow in Github that runs at midnight every day. It aims to check all microservices' master branches to detect policy validations in production code.

Bulk-Neodora workflow consists of five steps:

1. Getting a list of repositories to scan.
2. Getting Github token.
3. Going over the list of repositories where for every repository, it clones it and runs Neodora.
4. Creating a final HTML report based on saved reports.
5. Archiving results.

The way of running Neodora repeatedly for every repository keeps it simple, but it comes at the cost of total runtime. Currently, this solution takes around 21 minutes to scan 884 repositories. Network requests are known to be a factor that increases runtimes. In addition to cloning, some rules make GET requests to fetch up-to-date information. This raised questions: could Bulk-Neodora be improved by running the rule checking simultaneously for multiple repositories, and exactly what takes the most time for execution and how to reduce it?

5 Analyzing rule execution

To speed up the Bulk-Neodora workflow, it was decided to investigate where most of the time is spent during its execution. It was expected that repository cloning and GET requests inside the rules were why the workflow took so long. Cloning on repositories was already done with depth 1. This way, we only get a branch we are interested in and only the latest commit. Requests in the rules, however, can take some time to get a response, especially when the server is slow to answer.

5.1 Changes to capture rule runtimes

Some rules have GET requests (shown in listing 14) and do not use already stored data because some of the accepted values that the rule checks are expected to change. Having hard-coded data for tests to use would cause the need for many pull requests to keep the rules up to date.

```
node_release_date_request = http.send({
  "url": sprintf("https://endoflife.date/api/nodejs/%s.json", [
    get_nvmrc_version]),
  "method": "get",
  "raise_error": false,
  "cache": true,
  "timeout": "10s",
  "max_retry_attempts": 3,
})
```

Listing 14. GET request inside rule.

In this GET request example, the *nvmrc* value can vary from service to service. In addition to *nvmrc* value differences, many URLs contain a different service name. However, many URLs are completely the same, no matter what service. It was decided to record how long it takes to get the responses for the requests made in the rules.

Getting the start and end times of these requests in Rego was impossible. Although Open Policy Agent can be easily extended with custom built-in functions [13]. This means we can create a callable custom function to make a desired request and return the response while capturing the duration.

```
func registerBuiltinImpl1(bctx rego.BuiltinContext, a *ast.Term
    ) (*ast.Term, error) {
  var request map[string]interface{}
  var elapsed time.Duration
  var resp *http.Response
  var err error

```

```go
  if err := ast.As(a.Value, &request); err != nil {
    return nil, err
  }

  authToken := request["auth"]
  url := request["url"].(string)
  ruleName := request["rule_name"].(string)
  client := &http.Client{}
  req, err := http.NewRequest("GET", url, nil)
  req.Header.Add("Authorization", "token "+authToken.(string))

  if authToken.(string) != "" {
    start := time.Now()
    resp, err = client.Do(req)
    elapsed = time.Since(start)
    defer resp.Body.Close()
  } else {
    start := time.Now()
    resp, err = http.Get(url)
    elapsed = time.Since(start)
    defer resp.Body.Close()
  }
  runtimeResult := RuntimeResult{RuntimeType: "GET", RuleName: ruleName, RequestTo: url, Duration: elapsed.Microseconds()}
  writeResultToFile(runtimeResult)

  if err != nil || resp.StatusCode != http.StatusOK{
    return nil, err
  }
  respBody, err := io.ReadAll(resp.Body)
  if err != nil {
    return nil, err
  }

  var bodyAsJson any
  json.Unmarshal(respBody, &bodyAsJson)
  var interfaceData = ResponseResult{StatusCode: resp.StatusCode, Body: bodyAsJson}
  var responseAsValue, _ = ast.InterfaceToValue(interfaceData)
  var asterm = &ast.Term{Value: responseAsValue}
  return ast.ArrayTerm(asterm), err
}
```

Listing 15. GET request as a custom function.

Listing 15 shows an implementation part of the custom function. It converts given Rego values to Go values, returning early in case of an error (lines 7-9), and sets up data for a GET request (lines 11-16). A GitHub authentication token is sent with a request for GitHub API requests. Requests are timed, and the time taken is saved for future analysis (lines 29-30). Lastly, the request's response is converted to a suitable format for the Rego language. Duration is measured in microseconds. For every repository, it saves the runtime results in a new file to avoid locking.

5.2 Runtime results

A custom function allowed gathering runtimes for every rule execution. An example of a runtime result is shown in listing 16.

```
{
    "RuntimeType": "GET",
    "RuleName": "checkNextNodeRelease",
    "RequestTo": "https://endoflife.date/api/nodejs/18.json",
    "Duration": 116308
  },
```

Listing 16. Runtime result.

In addition, the full execution time of every single rule was also captured. Data analyzing was done in Python, where all the runtimes were aggregated to a single data frame shown in table 2.

Table 2. Rule runtimes.

i	Type	Rule name	URL	Duration (ms)
0	GET	sonarConfiguration	URL	130924
1	GET	namespace	URL	133442
2	GET	checkNextNodeRelease	URL	215748
...	...	...	...	...
57396	FULL	helmCompanyTemplateVersionEOL	URL	340844
57397	FULL	sonarScanBranchProtection	URL	378100
57398	FULL	helmChartVersion	URL	438613

In a table 2, the URLs are redacted as they can be long. Entries with a type GET represent only the request time, and entries with FULL are for the total rule execution time. This way, summing up all the GET and FULL runtimes is possible to see how much time is spent fetching data.

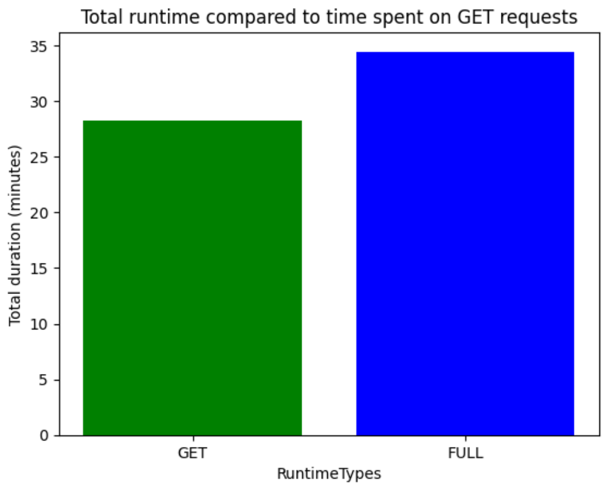

Figure 2. Time spent on GET requests & full rule execution time.

Figure 2 shows that 82% of the rules evaluation was spent on GET requests. After further analysis of URLs where requests are made, it was clear that some requests' responses could be used for other rule validations. From repository to repository, most URLs differed in repository name, version, and key. Still, there were also so-called static URLs that always returned the same data no matter what repository was being checked.

Before the overhaul of Neodora, it was also decided to test how much faster the tests run if there is no need for GET requests. It was done by gathering all the required data and using it as static values for the rules.

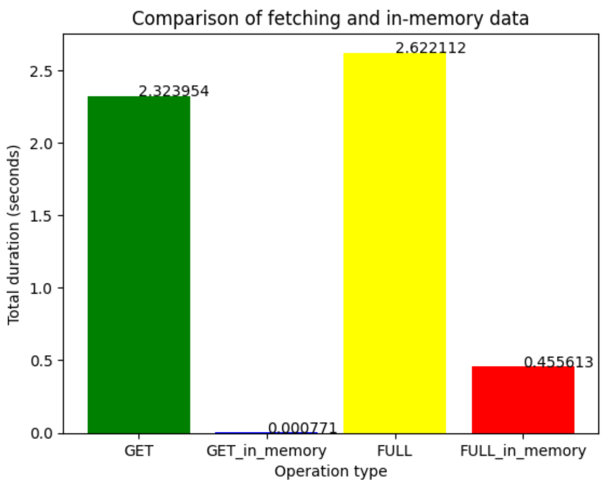

Figure 3. Difference in repository scan, with and without GET requests.

Figure 3 shows the gained difference in data fetching and full runtime for one repository. The data fetching was reduced to basically zero since the required data was already in memory. The total runtime was reduced nearly 6 times. It further proved that a great performance increase is expected for Bulk-Neodora just by having some GET request responses already in memory for rule evaluations.

6 The solution

During the implementation of the faster Bulk-Neodora, there was a lot of investigation and testing of different possible solutions. The most noteworthy are the ways of gathering the required repository files and getting around hardware and network limitations. The flow of operations is shown in figure 4.

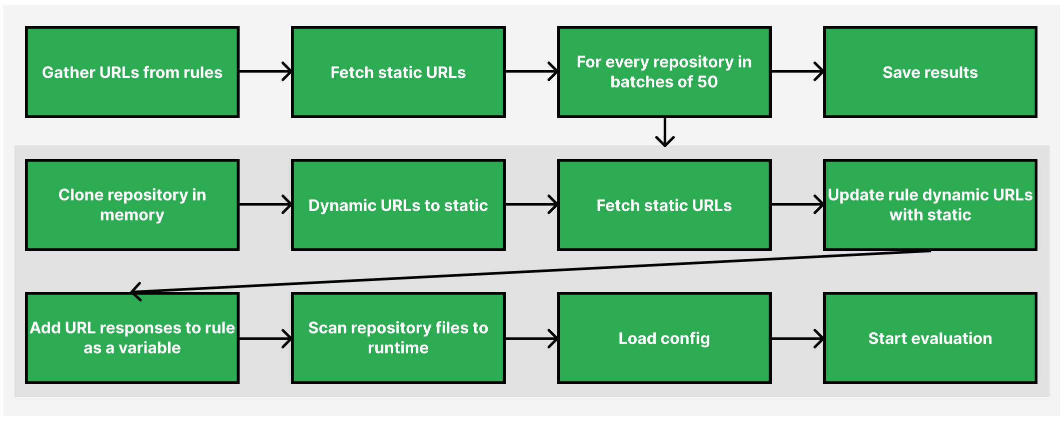
Gather URLs from rules
Fetch static URLs
For every repository in batches of 50
Save results
Clone repository in memory
Dynamic URLs to static
Fetch static URLs
Update rule dynamic URLs with static
Add URL responses to rule as a variable
Scan repository files to runtime
Load config
Start evaluation

Figure 4. Flow of operations.

The operation chain on a darker background runs for multiple repositories simultaneously. Waiting for another repository scan to finish is eliminated. URL requests that all scans need are made once beforehand.

6.1 Implementation

The high number of repositories in Bulk-Neodora scan made it necessary to change Neodora's logic to parallelise repository scans. The main idea was to make all the necessary GET requests before the rules were evaluated. Previously, URLs were inside rules, where GET requests were made as shown in listing 17.

```
request = response {
  response := http.send({
    "method": "get",
    "url": "https://missions.company.tools/api/tribes",
    "raise_error": false,
    "cache": true,
    "timeout": "10s",
    "max_retry_attempts": 3,
  })
}
```

Listing 17. GET request in rule.

By adding URLs to rule configurations, shown on a listing 18 on line 9, it was possible to get these at the start of Neodora execution. All the URLs from the rules were gathered and filtered based on whether they were static or dynamic.

```
1  rule.New(policy).
2      Id("tribeName").
3      Warning().
4      Name("Unknown tribe name").
5      Message("The property `owner` in `repository.json` must be
     a valid tribe name present in **mission-tracking**").
6      Documentation(doc).
7      Active().
8      Input("repository.json").AsJSON().
9      AddVariable("url0", "https://missions.company.tools/api/
     tribes").
10     OutputSchema(func(s *schema.Schema) {
11       s.Var("file_link")
12       s.Prop("file").Title("Repository file").RenderInBlock().
     AsLink("file_link")
13       s.Prop("status").Title("Owner status").RenderInTable().
     AsText()
14     })
```

Listing 18. Rule configuration.

An example of a static URL is in a listing 18. URL is considered dynamic when some part of it changes depending on a repository. A simple example would be *https://api.github.com/repos/org/<REPO_NAME>/contents/repository.json* where variable *<REPO_NAME>* is a corresponding repository name. Static URLs were used to gather data that every repository scan could use. After that, a list of repositories to scan is fetched, and a single Go goroutine is created for every repository scan.

It was decided that the cloning would be done in memory instead of cloning repositories to a file system to speed up the program even more. It was achieved using go-git [14], Git implementation in Go. Its goal is to be fully compatible with Git [15]. While, as of writing this paper, not all features are supported [16], the features we were interested in were implemented. More precisely, in-memory cloning and repository traversal. Repositories were cloned with depth 1, and the file contents were gathered by traversing a tree. Some variable values in dynamic URLs depended on values in the repository files. Once repository files were gathered, repository name, Node version, and Sonar project key variables in dynamic URLs were given their values. It enabled fetching responses for previously called dynamic URLs.

URL variables in rule configurations were then overwritten to be specific for a repository by replacing dynamic URLs with corresponding static ones. This allowed accessing URL responses from a map variable that was also added to the rule. The map variable had URLs as keys and responses as values. Rules to exclude from scans were filtered out, and the in-memory repository was scanned. The scanning part had to be rewritten because previously, it used the files on disk to parse them. It included finding all the required files for rules and then parsing them to a suitable format. The found files are cached, and empty files are classified as not found. Previous regex pattern support was kept, making it possible to find all the files in some directory that can exist anywhere in the repository. A repository-specific Neodora configuration file was read, and the required rules were updated. After that, a regular OPA evaluation started, and results for every repository were saved.

Improved Bulk-Neodora and the current in-use version were run on the same machine to compare runtime and output. Tests were run on MacBook Pro 2019 with a 2,6 GHz 6-Core Intel Core i7 processor. The network download speed averaged around 850 Mbps. Tests contained 881 repositories, as three left-out repositories required access permissions that the used GitHub token did not have. As the output of both versions was a JSON document, it was easy to do a difference check on output files. Both versions gave the same output. As of the runtime, for 881 repositories, it previously took 32 minutes. With the improved version, it was reduced to 2,2 minutes. Reducing the runtime almost fifteen times.

6.2 Blockers

During the improvement of Bulk-Neodora, multiple approaches were tried. Here are two main blockers that were encountered during the development.

As not all the repository files are needed for Neodora rule checking, the question of whether we could gather only the required files was raised. Git cloning does not support specifying files or folders to fetch, so the other option was to use GitHub API. GitHub API has an endpoint [17] that could be used for it. It can be used to get the contents of the files and also the contents of a directory, which would be sufficient. However, this API has a restriction that makes it unsuitable for Bulk-Neodora. GitHub API has rate limits [18]. Unauthenticated requests are limited to 60 per hour, and the limit is increased to 5000 for authenticated users. As the repository count and files needed for the Bulk-Neodora run are very high, the API rate limit was reached fast, and the run failed. It could be solved by having multiple tokens for authentication and/or using GitHub Enterprise Cloud. However, it was not tested as it would involve additional logic, and many small requests could take more time than one clone operation. Using GitHub API could be an option if the repository and unique file count are low.

Running many Neodora runs concurrently uses more system resources and has higher network requirements. In the development phase, it was discovered that while running Bulk-Neodora for 100 repositories finished, the machine sometimes turned unresponsive, and many GET requests experienced time-outs. It was clear that there had to be some throttling system. The solution was to limit the maximum concurrent scans to 50 and add retries to GET requests and cloning in case they time out. Exponential back-off was used for retry attempts. The necessary throttling and timeout lengths depend on the system and network, so different timeout lengths were tested. The maximum concurrent repository scans were set to 50 for the system's stability. When a scan finished, it freed up a spot for another.

7 Conclusion

The weak points of Rego rules implementation and Bulk-Neodora flow were investigated. Neodora runs for repositories were parallelised to cut down on the whole runtime, and reading data from a disk was replaced with in-memory operations.

This work introduced Datalog, Open Policy Agent, and Rego, gave a short history of Dora and Neodora, and highlighted their disadvantages. The findings suggest that while Rego is a fast and clear language for writing rules, its performance of rule evaluation can greatly suffer when it is waiting to receive data. The main operations that reduced the runtime of Bulk-Neodora were:

Replacing default cloning with in-memory cloning. Reading and writing to a disk are much slower than memory operations. If the hardware resources allow it, keeping all the data in memory improves the execution time.

Removing HTTP requests from rules. Making the required HTTP requests beforehand and adding the responses to the rule configurations clears the code that is meant to be only validation logic. In addition to making the rule easier to debug, it also makes it possible to make these requests in bulk.

Using multiple threads to execute many runs simultaneously. Instead of running one repository scan after another, it is possible to run multiple at the same time. The repository scans are not dependent on each other, so we can speed up the process by allocating more system resources.

The improved version of Bulk-Neodora has not been deployed yet. Some refactoring is left to do. In addition, it is planned to make it more user-friendly and increase coverage of the new code. Nonetheless, applying the first Futamura projection can make the program more specialised and increase performance.

References

[1] Loide K., Carneiro Alves de Lima B. R., Jakovits P., and Demidov J. Using datalog for effective continuous integration policy evaluation. In *Software Quality as a Foundation for Security*, pages 41–52, Cham, 2024. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-56281-5 (11.04.2024).

[2] Warren D. S. Top-down and bottom-up evaluation procedurally integrated, 2018. https://arxiv.org/abs/1804.08443 (06.05.2024).

[3] Open policy agent: Introduction [internet], 2024. https://www.openpolicyagent.org/docs/latest/ (11.04.2024).

[4] Introduction to the go programming language [internet], 2024. https://www.codemag.com/Article/2011051/Introduction-to-the-Go-Programming-Language (08.05.2024).

[5] Open policy agent: Policy testing [internet], 2024. https://www.openpolicyagent.org/docs/latest/policy-testing/ (29.04.2024).

[6] Mogensen T. Æ. Partial evaluation: Concepts and applications. In *Partial Evaluation*, pages 1–19, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-47018-2_1 (26.04.2024).

[7] Jones N. D. Lecture 3 on partial evaluation: Binding times, optimal specialisation and types [internet], https://www.nii.ac.jp/userimg/lectures/neiljones/TokyoSlidesOptimality-2.pdf (03.05.2024).

[8] Bansal S. Col728: Compiler design, introduction to interpreters, compilers, and programming languages [internet], 2024. https://iitd.github.io/col728/lec/intro.html (29.04.2024).

[9] Jones N. D., Gomard C. K., and Sestoft P. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., USA, 1993. https://www.researchgate.net/publication/213882771_Partial_Evaluation_and_Automatic_Program_Generation (01.05.2024).

[10] Gozillon A., Haeri S. H., Riordan J., and Keir P. Performance analysis of compiler support for parallel evaluation of c++ constant expressions. In *Software, System, and Service Engineering*, pages 129–152, Cham, 2024. Springer Nature Switzerland. http://dx.doi.org/10.1007/978-3-031-51075-5_6 (04.05.2024).

[11] Jordan H., Scholz B., and Subotić P. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing. https://doi.org/10.1007/978-3-319-41540-6_23 (02.05.2024).

[12] Paljasma T. Validating docker image and container security using best practices and company policies, Tallinn University of Technology Master's thesis, May 2019. https://digikogu.taltech.ee/et/Item/b9367be6-0646-4c2f-b32b-56ee8a024f0d (13.04.2024).

[13] Open policy agent: Extending opa [internet], 2024. https://www.openpolicyagent.org/docs/latest/extensions/ (12.04.2024).

[14] go-git: A highly extensible git implementation in pure go [internet], 2024. https://github.com/go-git/go-git (15.04.2024).

[15] Git: Git source code mirror [internet], 2024. https://github.com/git/git (15.04.2024).

[16] go-git: A highly extensible git implementation in pure go | compatibility [internet], 2024. https://github.com/go-git/go-git/blob/master/COMPATIBILITY.md (15.04.2024).

[17] Rest api endpoints for repository contents - github docs [internet], 2024. https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28 (15.04.2024).

[18] Rate limits for the rest api - github docs [internet], 2024. https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28 (15.04.2024).

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public