ArchLintor: detecting architectural violations in TypeScript systems

Matheus Resende
Depto de Ciência da Computação
Universidade Federal de Lavras
Lavras, Brasil
matheusresende.comp@gmail.com

Rafael Durelli
Depto de Ciência da Computação
Universidade Federal de Lavras
Lavras, Brasil
rafael.durelli@ufla.br

Ricardo Terra
Depto de Ciência da Computação
Universidade Federal de Lavras
Lavras, Brasil
terra@ufla.br

ABSTRACT

Maintaining architectural integrity is challenging in fast-paced development environments, especially in TypeScript systems where architectural analysis is hindered by language complexity. This paper introduces `ArchLintor`, a static analysis tool for architecture conformance checking in TypeScript projects. By allowing architects to define rules over module interactions, `ArchLintor` detects architectural violations, classifies them (e.g., divergences, absences, alerts), and produces detailed reports and visualizations through graphs and dependency structure matrices (DSM). The tool supports CI/CD integration, serving as a quality gate to prevent architectural erosion before code is merged. Entirely built from scratch using widely adopted JavaScript ecosystem tools, `ArchLintor` offers developers a practical solution to monitor and enforce architectural rules continuously.

Demo video: https://doi.org/10.6084/m9.figshare.29123549

KEYWORDS

Architecture conformance checking, TypeScript, Static Analysis.

1 Introduction

Defining architecture is a common step in the software lifecycle, establishing rules and standards that the development team must follow throughout subsequent steps [29]. This step is crucial for ensuring the healthy and appropriate growth of software, avoiding the emergence of technical debt, simplifying test creation, and debugging processes since the communication structure of the interfaces would be predefined [28]. The technology market's heating up, increasing development team turnover, the race for time to market, and the difference in technical and business knowledge among team members accelerate the process of architectural erosion, characterized by the difference between implemented and planned architecture [25].

With the popularization of the DevOps culture, software developers have become more engaged with quality processes, testing, and distributing produced software [9]. This movement creates a demand for tools capable of validating and formatting code against predefined standards. The receptive market for new tools that contribute to code quality has led to the idea of developing tools for architectural compliance analysis.

JavaScript is currently the second most used language [33]. Its popularity comes from the ability to be used both on the client and server sides and the ease of developing using supersets and variations that are transpiled to pure JavaScript [5]. TypeScript and React are two examples of this. React uses a JavaScript variation that allows inserting HTML code alongside JavaScript code, known as JSX (JavaScript Syntax Extension) for JavaScript and TSX (TypeScript Syntax Extension) for TypeScript. TypeScript is a superset that adds type annotation capabilities, converting to native JavaScript using a compilation process [3, 5].

Despite the benefits, the increasing complexity and feature-rich nature of JavaScript and TypeScript present challenges in performing effective architectural analysis. There is a noticeable gap in tools specifically designed for architectural compliance in these languages, which can address architectural erosion by identifying and correcting deviations before they become ingrained in the system.

This work proposes the creation of a static code analysis tool aimed at identifying architectural deviations from the source code and a high-level description of the proposed system architecture. The tool defines the modules and their relationships in terms of permission (allowed and forbidden) and obligation (required). It provides a textual report pointing out each architectural deviation and generates a directed graph [22] and a DSM (Dependency Structure Matrix) [30] highlighting the found architectural deviations.

The tool aims to maintain software system architecture by detecting deviations before integration, allowing corrections before causing significant impacts and thus reducing correction costs [20]. Furthermore, it presents opportunities to refine the current architectural design and could be integrated into CI/CD pipelines, serving as a quality gate to prevent the integration of changes that violate architectural constraints [23].

This paper is structured as follows: In Section 2, we present related work. Section 3 introduces the `ArchLintor` tool. In Section 4, we briefly describe the project's implementation. Section 5 offers a discussion on the findings and the tool's capability of identifying architectural deviations in a project from a set of rules and generating artifacts – JSON, graph, and DSM. Finally, Section 6 concludes.

2 Related Work

TypeScript, as one of the most widely used programming languages according to GitHub [33], has a community that develops various support tools to address diverse problems. Among these tools are static analyzers responsible for code quality and security, visualization generators, etc. Our `ArchLintor`, as a tool that performs static code analysis with a focus on quality through the identification of architectural deviations, resembles other available tools in the market, forming part of the language's ecosystem.

Among static analyzers, language servers are prominent, widely used in text editors, and tools related to style and code quality [19]. Language servers provide text editors with features such as code suggestions, shortcuts to documentation and implementation, and the identification of syntax and logic errors in certain cases [2, 4].

Typically, language servers run during the coding process, identifying identifier-related errors at write-time and more complex problems at save-time. Examples of language servers are TypeScript Language Server for TypeScript and JavaScript [17], and Quick Lint [15] and Flow [13] for JavaScript.

Other static analysis tools are related to quality and security, responsible for finding bugs, code smells, and vulnerabilities [19]. These tools can be configured to run after or during the source code save, during a commit to the version manager, and also in CI/CD flows [23]. Quality tools focused on style, for example, can make changes to the source code to fit style rules, such as removing or adding semicolons at the end of lines, homogenizing indentation characters, etc. Prettier is an example of an ecosystem tool that performs aesthetic checks and can also apply corresponding corrections [6, 14]. Complementarily, ESLint is a lightweight analysis tool that points out possible bugs, syntax errors, and other functionalities added via plug-in, such as security analyses [12]. SonarQube [11], in turn, is a more robust and multilingual tool capable of performing a more complete and in-depth analysis, identifying problems that ESLint cannot detect.

Visualization tools allow for the planning and understanding of the software at that moment. Graphs, UML diagrams, and DSM are some possible visualization forms to extract. An example of a visualization tool is Arkit, which, from the source code, produces dependency graphs based on the schema file that declares the architectural components [1]. For UML diagrams, the Tplant tool adapts the TypeScript code to the PlantUML format used for diagram rendering [16]. JetBrains IDEs, among the resources and tools for development such as code completion and code generation, also offer code visualization features, displaying hierarchical project structures and providing information about class and interface hierarchies [7]

This work is based on ArchRuby [26] and ArchPython [21], which proposed, respectively, similar tools for Ruby and Python projects. These tools perform static code analysis, identifying dependencies between code components and relating them to the specified ideal architecture. These tools also generate graph and DSM visualizations, representing the relationships between defined modules and highlighting those with architectural deviations. Another tool that inspired this work was DCLsuite, which is based on the Dependency Constraint Language (DCL), a language specialized in rule definitions. DCL offers broader specification resources than the simple module and file definition proposed in this work [31]. Based on DCL, DCLcheck also performs architectural compliance analysis for Java projects that have their architecture described using DCL. DCLCheck is an Eclipse IDE plug-in, allowing architectural deviations to be pointed out in the editor itself [32]. It is important to note that `ArchLintor` was carefully designed to leverage technologies and libraries widely adopted by the JavaScript community. Therefore, our solution and implementation were developed entirely from scratch.

The `ArchLintor` tool differs from ArchRuby, ArchPython, and DCLSuite in the characteristics of the treated language. On one hand, Java is a strongly typed language with compile-time type checking. On the other hand, Python and Ruby are weakly typed languages, where type annotation is entirely optional and unnecessary for code execution, which greatly complicates compliance analysis due to the lack of explicit types. TypeScript using the strict compilation mode forces the developer to explicitly use type annotations. Even using strict mode, certain constructions that require inference are still allowed, such as variable initialization where the variable type takes the type of the assigned value.

3 ArchLintor

The proposed solution is based on a static analyzer implemented in TypeScript, capable of identifying architectural deviations by parsing the source code. Its objective is to provide development teams with a way to monitor architectural erosion through reports and graphical representations. Figure 1 represents the sequence relationship between the processing stages:

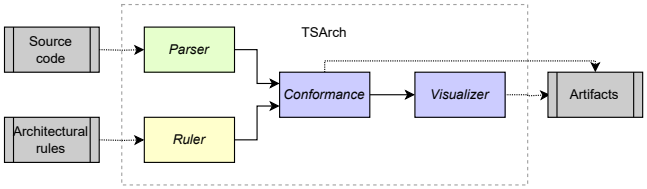

Figure 1: Overview of `ArchLintor`

(1) Parsing of the source code, populating the symbol table (Parser);
(2) Parsing of the rule set (Ruler);
(3) Conformance analysis (Conformance); and
(4) Visualization of the architecture and its deviations (Visualizer).

Software engineers may adopt ArchLintor at any stage of the development process. A recommended practice is to maintain the architectural rule specifications at the root of the repository and encourage developers to execute the tool locally prior to each commit. When violations are detected, developers are expected to investigate the cause and address the underlying issue. In rare cases, the violation may indicate a need to refine the architectural rules themselves to better reflect the system's intended architecture.

At a more advanced level of adoption, the artifacts produced by our approach can be incorporated into CI/CD pipelines, where they may serve as quality gates—blocking code integrations that violate architectural conformance [23]—even though integration itself is not the primary focus of this paper. This enables the early detection of architecture-related violations during development, helping to reduce the cost of corrective actions [20]. An additional strength of the tool lies in its ability to visualize the system's architecture, effectively highlighting areas affected by architectural erosion.

3.1 Running example

In this section, we illustrate our approach using a project named `Corrector.ts`, developed by the first author of this work. This project is a simple application responsible for grading exams and generating statistics on the obtained results. We use this running example to demonstrate the feasibility of our approach and to provide a concrete guide for understanding each step of the proposed solution.

The command line receives the path to a file containing a list of students and their respective answers, as well as another path to a file containing the answers to the exam in question. After that, based on the results, statistics are calculated. Finally, the data is persisted in a "S3 Bucket". The planned architecture for this project consists of eight modules following a non-strictly layered strategy:

- `Model`: Data structures for persistence and manipulation.
- `Service`: Adapters that act as intermediaries between external services and the business logic of the application.
- `Statistics`: Responsible for manipulating the provided data and producing new information.
- `Corrector`: A set of functions responsible for comparing exam answers with a given answer key.
- `Reader`: Responsible for interpreting the data contained in the file and converting it to the program's data structure.
- `CLI`: Responsible for identifying the format of the received file and its paths. It has a subdirectory called "Commands" where subcommands and parameters are specified.
- `AWS-S3`: An abstract module representing the external libraries used to communicate with the AWS storage service.
- `Util`: Auxiliary functions for date handling.

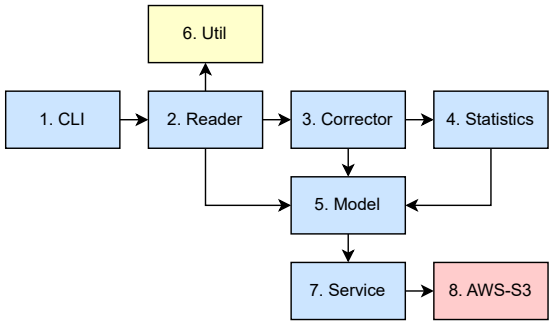

Figure 2: Architecture of `Corrector.ts`

Figure 2 illustrates the planned architecture of the application. The **CLI** module consists of files that define the possible commands, utilizing the Yargs[1] library. The **Reader** module consists of creating a appropriate file reader to convert to the models defined in the **Models** module. The **Corrector** is responsible for grading each exam question based on the answer key. After grading, the **Statistics** module calculates the metrics. The **Models** module, in addition to defining the used interfaces, is used for data persistence after all analyses. The **Services** module provides implementations of support functions that act as adapters for external services. In the **Utils** module, there are functions not necessarily related to the project, in this context, related to date processing.

3.2 Parsing of the rules

The architecture specification is done through a JSON (JavaScript Object Notation) file. This choice was made precisely because it is a widely adopted format in the JavaScript community, as it is based on the language's own object notation [24]. Another reason for this choice is the practicality of building the specification, as it is a structured, easy-to-read text file that does not require other computational resources and is simpler to write compared to markup languages such as XML (Extensible Markup Language).

As seen in Listing 1, the file consists of an anonymous global object that contains the definition of the modules. Each module has a name (e.g., `"ModuleID"` in line 2) and can be defined by the attribute `"files"` or `"package"` (lines 3 and 4). The architectural constraints are defined in attributes `"allowed"`, `"forbidden"`, and `"required"` (lines 5 and 6), which would contain the files or modules that, respectively, you can, cannot or must establish dependency with.

Listing 1: Example JSON configuration

```
1 {
2   "ModuleID": {
3     "files": ["Files"],
4     "packages": ["External libraries"],
5     "forbidden" or "allowed": ["Files or modules"],
6     "required": ["Files or modules"] #optional
7   }
8 }
```

Given the lack of a module system, the syntax of the rules directly considers the arrangement of files in the directory structure to simplify the specification. It is strongly recommended that the defined modules do not use only part of the files contained in the directory, but rather all of them, taking advantage of the wildcards marked by the symbols `*` and `**`.

For example, consider a directory named `foo` containing the files `./foo/a.ts`, `./foo/b.ts`, `./foo/bar/c.ts`, and `./foo/bar/d.ts`, where the latter two are located within a subdirectory bar. A rule defined as `./foo/*` applies only to the files directly within the `foo` directory, excluding any files in its subdirectories. In contrast, the rule `./foo/**` includes all files within foo and its entire directory tree. Therefore, under the rule `./foo/*`, only `a.ts` and `b.ts` are included in the declared module, while the rule `./foo/**` additionally encompasses `bar/c.ts` and `bar/d.ts`.

The `package` field, in turn, does not consider its subdivisions. If a library is subdivided into other parts, only what is explicitly declared is considered. For example, in line 32 of Listing 2, only `"@aws-sdk/client-s3"` will be considered. For instance, `"@aws-sdk/client-s3/foo"`, would not belong to the module. For simplification purposes, all external libraries not declared in a module—by default—can be used in any part of the code.

The `allowed` and `forbidden` fields are mutually exclusive and therefore cannot be used simultaneously. When a module M is defined without any rules, the tool assumes that M is not allowed to establish any dependencies. At this stage, adding an `"allowed"` constraint modifies the semantics to explicitly permit dependencies on specific modules. Conversely, when a `"forbidden"` constraint is defined, it permits dependencies on all modules except those explicitly listed. When a module, file, or library appears in the `required` field, it automatically implies being allowed. It is important to mention that the rule parser does not validate the coherence of the specification, so a poorly made specification can lead to inconclusive results.

[1]Yargs is a library that assists in creating command-line tools by simplifying the parsing of received arguments [10].

Listing 2: Specification of Corrector.ts

```
1  {
2    "CLI": {
3      "files": ["./src/cli/**"],
4      "allowed": [],
5      "required": ["Reader"]
6    },
7    "Corrector": {
8      "files": ["./src/corrector/*"],
9      "forbidden": ["CLI", "AWS-S3"]
10   },
11   "Statistics": {
12     "files": ["./src/statistics/*"],
13     "allowed": ["Model"]
14   },
15   "Reader": {
16     "files": ["./src/reader/*"],
17     "allowed": ["Model", "Util"],
18     "required": ["Corrector"]
19   },
20   "Model": {
21     "files": ["./src/model/*"]
22   },
23   "Util": {
24     "allowed": [],
25     "files": ["./src/util/**"]
26   },
27   "Service": {
28     "files": ["./src/service/**"],
29     "allowed": ["AWS-S3"]
30   },
31   "AWS-S3": {
32     "packages": ["@aws-sdk/client-s3"]
33   }
34 }
```

Listing 2 deals with the architecture specification of the example described in Section 3.1 using the structure previously described. As can be observed, the **Models** module defined in lines 20 to 22 does not have `allowed` and `forbidden` rules, so the tool interprets it as everything forbidden. The **CLI** module defined in lines 2 to 6 has subdirectories, and it is reasonable for the application context to consider all other files in the subdirectories. The other modules do not have subdirectories, so in this specific scenario, the use of `*` or `**` is indifferent. Although indifferent, `*` is used as it is more restrictive in case new directories are created and require some architectural evolution.

3.3 Parsing of the source code

This part of the tool is responsible for finding dependency markers, highlighting all the necessary information for conformance analysis. For this, the syntax tree obtained by compiling the code is traversed, searching for "import" structures, type declarations, and variables.

Since TypeScript does not require type declarations in all contexts, in this work, type inference is also performed using type-checking structures obtained by compiling the code with the `strict` option enabled. This ensures that the code adheres to the strictest typing rules and is a *prerequisite* for our tool's analysis. For example, when declaring a function, its parameters must explicitly declare the type `any` if it is allowed to receive any type as that parameter. Another example is related to the use of `undefined` and `null`, which must respect their own types.

The intrinsic complexity of the TypeScript language led us to narrow the current scope of the tool. Consequently, features related to programming paradigms other than imperative or object-oriented are not currently supported. That is, the analysis is limited to type declarations (such as classes, interfaces, enumerations, and type aliases), object structures (including attributes, methods, functions, and variables), and import statements.

3.4 Conformance checking

With the results of parsing the rules and the source code, a comparison can then be made between the dependencies allowed in the rules file and those actually used in the project, identifying the following cases [27]:

- **Convergence**: When a file from module `A` depends on a file from module `B` directly or indirectly and (i) there is a rule that allows this dependency or (ii) there is no explicit rule prohibiting the dependency.
- **Divergence**: When a file from module `A` depends on a file from module `B` directly or indirectly, (i) but there is no rule allowing this dependency or (ii) it is explicitly prohibited.
- **Absence**: There is a file from module `A` that has no dependency on a file from module `B`, even though there is a rule requiring `A` to depend on `B`.
- **Alert**: There is a rule that allows (but does not require) files from module `A` to depend on files from module `B`, but no file from `A` depends on any file from `B`. This is not a deviation but may point to specification flaws.

We intentionally inserted the following four violations into the example defined in Listing 2 to simulate possible real-world situations:

- **Absence**: A new file is created in the `CLI` module with the aim of defining a command for reading JSON files. However, this file does not use a reader defined in the `Reader` module to process the provided JSON file, breaking the rule that `CLI` must depend on `Reader`. Line 5 of Listing 2 specifies this rule.
- **Divergence**: During the creation of the new command for reading JSON files, the "`Exam`" model is imported, breaking the rule that prohibits a file from the `CLI` module from depending on a file from the `Model` module. Line 4 of Listing 2 specifies all the modules whose dependency is allowed, but `Model` is not on the list.
- **Divergence**: During the correction, an intermediate result is produced which is persisted in an "S3 Bucket". The correction is performed in the `Corrector` module and the persistence is done using a library described in the `AWS-S3` module, which is explicitly prohibited in line 9 of Listing 2.
- **Alert**: Some functions are commonly used in various modules and do not represent business rules of the application, so they were extracted to the `Util` module. The `Reader` module allows the use of `Util`, as described in line 17 of Listing 2, but the implementation does not use it, which constitutes an alert.

As can be observed in Listing 3, the result of the architectural conformance checking points outs to occurrences of divergences, absences, and alerts. Due to space constraints and since the report of divergences are similar, we omit the second divergence from `Corrector` to `AWS-S3` module.

Listing 3: Subset of ArchLintor's result on Corrector.ts

```
{
  "ABSENCES": [
    {
      "targetFile": "./tool/src/cli/command/json.ts",
      "originModule": "CLI",
      "targetModule": "Reader"
    }
  ],
  "DIVERGENCES": [
    {
      "line": 2,
      "kind": "importation",
      "originFile": "./tool/src/cli/command/json.ts",
      "targetFile": "./tool/src/model/exam.ts",
      "originModule": "CLI",
      "targetModule": "Model"
    },
    ...
    ],
  "ALERTS": [
    {
      "originModule": "Reader",
      "targetModule": "Util"
    }
  ]
}
```

3.5 Results and visualization

As previously described, the result of the conformance checking is a textual report similar to the one presented in Listing 3. The JSON report format allows for the integration of the tool with other software systems, in addition to allowing easy human interpretation and presenting details such as the file and the line of code that produced the deviation. Nevertheless, in large systems that present many deviations, interpreting the textual result can be slow and somewhat confusing. Therefore, visual results are also produced. The visual representations, in addition to displaying the points of architecture erosion, also show the communication of all application modules. The greatest advantage of visual representations is the rapid identification of erosion points through the symbols and colors of the edges and matrix cells.

In the graph, the nodes represent the modules and the directed edges represent the dependencies [22]. The direction of the arrow represents the direction of the dependency, the number followed by the **"#"** symbol represents the number of occurrences of that dependency, and dashed edges represent architectural deviations. As can be seen in Figure 3, the `Statistics` module depends on the `Model` module. The orange and red colors and the symbols **"!"** and **"X"** respectively highlight divergences and absences in the image. Alerts, in turn, are represented by the color gray and the symbol **"?"**.

Another form of visualization is the DSM (Dependency Structure Matrix) used to scalably represent the dependencies of a project [30]. In this implementation, the columns represent the modules from which the dependencies originate, while the rows represent the modules with which the dependencies are established. Thus, the reading should be "column module depends on row module". For example, it can be seen in Table 1 that `Statistics` (column number 3) depends on `Model` (row number 1) eleven times. DSM follows the same color and symbol logic as the graph.

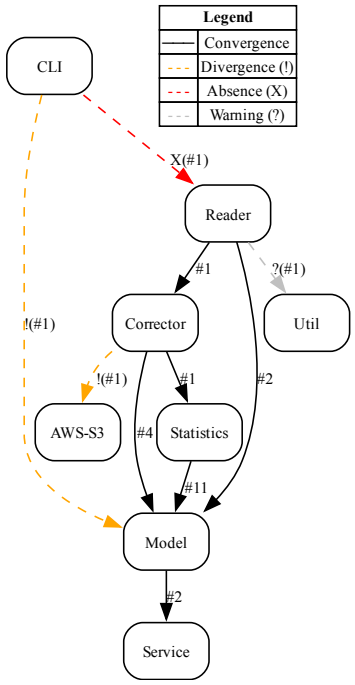

Figure 3: ArchLintor's graph result on Corrector.ts

Table 1: ArchLintor's DSM result on Corrector.ts

Modules	1	2	3	4	5	6	7	8
1 - Model			11	4	2	!1		
2 - Service	2							
3 - Statistics				1				
4 - Corrector					1			
5 - Reader						X1		
6 - CLI								
7 - AWS-S3				!1				
8 - Util					?1			

4 Project and Implementation

The tool was organized into five modules, each representing a stage of its execution: Data Input (CLI), Rule Processing (Ruler), Source Code Processing (Parser), Conformance Checking (conformance), and Artifact Creation (visualizer).

CLI: This module collects the paths for the source code and the rules, as well as the list of files belonging to the project. This information is provided to the subsequent modules. The `ArchLintor` tool is designed as a command-line interface (CLI), which was developed using Yargs that provides all the necessary features for capturing information, arguments, and assistance. The execution of the tool involves some parameters but the only required one is the project path.

Ruler: This module processes the rules, simplifying their format to be used in the compliance analysis based on the rule set. When handling the `"files"` and there are symbols `*` or `**`, this module replaces this syntax with a list of files. For instance, `"foo/*"` would

be the actual list of files that it represents (e.g., `"a,b,c,d"`). By definition, a module can have `"allowed"` or `"forbidden"` rules and one set is necessarily disjoint from the other. In our design, since we chose to use only `"allowed"`, `"forbidden"` rules are converted to `"allowed"`. Assuming a system that has four files namely a, b, c, and d, when you define `"forbidden"="a"`, this module replaces it with `"allowed"="b,c,d"`.

Parser: This module links language elements—e.g., variables, classes, interfaces, functions, and parameters—to their types and, consequently, to the file that implements these elements, defining a *dependency*. This is accomplished through a comprehensive list that includes each element's type, source file, and line number. The TypeScript Compiler API plays a crucial role in this process by enabling navigation through the code structure and extraction of dependency-indicative information, such as the syntax tree and TypeChecker [8]. Type inference is a key feature utilized here, where the compiler assigns types based on variable assignments and array specifications, enhancing the accuracy of dependency detection. For instance, type inference allows for the deduction of types through assignments and the characteristics of array elements, ensuring that only necessary types are considered to avoid redundancy. Furthermore, this module leverages TypeScript's ability to handle types without explicit declarations, which adds complexity in identifying dependencies solely through imports. Qualified names are used to resolve ambiguities and link types to their origins effectively, which is critical for recognizing implicit dependencies within the project's structure. However, this sophisticated identification process complicates the use of cache as it necessitates complete project compilation.

Conformance: Based on the simplified rules and the set of dependencies, this module performs the conformance checking, identifying architectural deviations and providing them to the visualizer. Each file is mapped to its respective module to assess dependencies against defined rules. Divergences are identified by checking if dependencies comply with the `allowed` list (there is no `forbidden` in this stage). Absences are identified by checking whether files indeed establish dependency with the `required` list. Although unused required files indicate absences, unused allowed files indicate alerts. Ultimately, this modules provides us with dependencies, divergences, absences, and alerts.

Visualizer: This module uses the information about dependencies and deviations to produce the artifacts, specially visual artifacts. Three main artifacts are generated: a textual report excluding convergences, and visual representations such as a graph and a Design Structure Matrix (DSM). These artifacts reflect all dependencies identified during the analysis. The process involves aggregating dependency data to understand relationships between modules, using tools like GraphViz [18] for visualization.

5 Discussion

Under the demand for improvements and new functionalities, software systems are subject to drifting away from their original architecture. This process, known as architectural erosion, can hinder problem resolution and the addition of new functionalities. Therefore, studies on architectural compliance help identify and prevent architectural erosion.

The TypeScript language, a superset of JavaScript, is currently one of the most widely used languages and offers developers great coding freedom. More importantly, it has a vast ecosystem of libraries, frameworks, and tools, as well as a community engaged in development and maintenance. In the context of compliance analysis, the freedom that the language offers poses a challenge for the development of analysis tools that cover the full scope of possible solutions such as Node and Deno ecosystems and syntax extensions like TSX implemented by the React library.

The findings of this study clearly show the proposed tool implements a solution capable of identifying architectural deviations in a project from a set of rules and generating artifacts – JSON, graph, and DSM – that represent the relationship between the modules and highlight the deviations found. Architectural deviations are identified based on the dependencies between the modules declared in the rules and are classified into divergences, absences, and alerts, which are highlighted differently in the visualizations: edges styled in different shapes, colors, and symbols in the graph, cells with distinct colors and symbols in the DSM, and separate lists in the textual report. One application of this tool is associated with continuous integration and delivery (CI/CD) flows, for example, as a quality gate in a stage before the application deployment to prevent deployment if any deviation is found.

6 Conclusion

This work addresses the challenges of architectural analysis in TypeScript by proposing `ArchLintor`, a tool designed to perform architecture conformance checking based on a set of rules specified by the architect. The methodology includes defining architectural rules and utilizing our proposed tool to ensure compliance with these rules.

Future work includes: (i) evaluating the tool using a real-world or open-source project to increase the credibility and representativeness of the results; (ii) implementing a more robust rule definition language that considers language features such as inheritance, naming conventions, and type declaration forms similar to what DCL proposes for the Java language; (iii) increasing the tool's analysis scope to cover more paradigms such as generic programming, covering syntax extensions like TSX, and supporting projects in the Deno ecosystem; and (iv) improving the produced visualizations to present, in addition to the relationship between modules, the similarity between modules. In this case, the suggestion is to use clustering techniques to generate images where similar modules or those with greater interaction are represented closer together both in the graph and the DSM.

ARTIFACT AVAILABILITY

The artifacts generated by `ArchLintor`—including JSON reports, dependency graphs, and DSM matrices—can be accessed and utilized to ensure continuous architectural compliance and facilitate further research and development on https://github.com/matheusHResende/archlintor.

ACKNOWLEDGMENTS

This work is supported by FAPEMIG grant APQ-03513-18.

References

[1] 2020. Visualises JavaScript, TypeScript and Flow codebases as meaningful and committable architecture diagrams. https://github.com/dyatko/arkit

[2] 2021. What is the Language Server Protocol? https://microsoft.github.io/language-server-protocol

[3] 2022. Introduzindo JSX. https://pt-br.reactjs.org/docs/introducing-jsx.html

[4] 2022. Langserver.org: A community-driven source of knowledge for Language Server Protocol implementations. https://microsoft.github.io/language-server-protocol/

[5] 2022. Modules. https://www.typescriptlang.org/docs/handbook/2/modules.html

[6] 2022. Prettier: Code style options. https://prettier.io/docs/en/options.html

[7] 2022. Source code hierarchy. https://www.jetbrains.com/help/webstorm/viewing-structure-and-hierarchy-of-the-source-code.html

[8] 2022. Using the compiler API. https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API

[9] 2022. What is DevOps? https://aws.amazon.com/pt/devops/what-is-devops/

[10] 2022. Yargs. https://github.com/yargs/yargs

[11] 2023. Clean code for teams and enterprises with SonarQube. https://www.sonarsource.com/products/sonarqube/

[12] 2023. Find and fix problems in your JavaScript code. https://eslint.org/

[13] 2023. Flow for VSCode. https://github.com/flow/flow-for-vscode

[14] 2023. Prettier code formatting. https://prettier.io/

[15] 2023. Quick Lint JS. https://github.com/quick-lint/quick-lint-js

[16] 2023. Tplant. https://github.com/bafolts/tplant

[17] 2023. TypeScript Language Server. https://github.com/typescript-language-server/typescript-language-server

[18] 2024. GraphViz. https://graphviz.org

[19] David Binkley. 2007. Source Code Analysis: A Road Map. In *2007 Future of Software Engineering (FOSE)*. 104–119.

[20] Bill Brykczynski, Reginald Meeson, and David A Wheeler. 1994. *Software inspection: Eliminating software defects*. Institute for Defense Analyses, Alexandria.

[21] Eduardo F de Lima and Ricardo Terra. 2020. ArchPython: architecture conformance checking for Python systems. In *34th Brazilian Symposium on Software Engineering (SBES)*. 772–777.

[22] John W Essam and Michael E Fisher. 1970. Some basic definitions in graph theory. *Reviews of Modern Physics* 42, 2 (1970), 271.

[23] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html

[24] Wallace Jackson. 2016. *JSON quick syntax reference*. Apress.

[25] Ruiyin Li, Peng Liang, Mohamed Soliman, and Paris Avgeriou. 2022. Understanding Software Architecture Erosion: A Systematic Mapping Study. *Journal of Software: Evolution and Process* 34 (2022), e2423.

[26] Sergio Miranda, Marco Tulio Valente, and Ricardo Terra. 2015. ArchRuby: Conformidade e Visualização Arquitetural em Linguagens Dinâmicas. In *VI Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session*. 17–24.

[27] Gail Murphy, David Notkin, and Kevin Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *3rd Symposium on Foundations of Software Engineering (FSE)*. 18–28.

[28] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *Software Engineering Notes* 17, 4 (1992), 40–52.

[29] S Shylesh. 2017. A study of software development life cycle process models. *14th National Conference on Reinventing Opportunities in Management, IT, and Social Sciences (MANEGMA)*, 534–541.

[30] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. 2001. The Structure and Value of Modularity in Software Design. *Software Engineering Notes* 26, 5 (2001), 99–108.

[31] Ricardo Terra and Marco Tulio Valente. 2008. Towards a Dependency Constraint Language to Manage Software Architectures. In *2nd European Conference on Software Architecture (ECSA)*. 256–263.

[32] Ricardo Terra, Marco Tulio Valente, and Luis Fernando Miranda. 2012. Conformação Arquitetural com DCLcheck. *MundoJ* X, 55 (2012), 44–49.

[33] Carlo Zaponi. 2022. GitHut 2.0. GitHub. https://madnight.github.io/githut/#/pull_requests/2022/1