Evaluating accessibility testing in automated software build processes

Aleksander Bai, Rannveig A. Skjerve, Till Halbach, Kristin Fuglerud

Abstract

Today, most software projects utilize an automated build process with unit tests, code quality checks, end-to-end integration tests, and more. To make software usable by as many people as possible, regardless of capabilities, accessibility testing must be integrated into the build process. The goal is highly accessible software where issues are found early in the development lifecycle. In this work, we have investigated the most common accessibility testing tools suitable for integration in an automated build process and split the tools into two categories based on their rulesets. The rulesets are evaluated against a well known demonstration site for calculation of the rulesets' precision, recall, and F_1 scores. Finally, we discuss the implications of their scores and how accessibility testing tools can be integrated into an automated build process.

Accessibility, Software Development, Testing, Automated Build Process.

1 INTRODUCTION

The aim of accessibility testing is to evaluate whether the final solution will be accessible for a wide range of people, including people with various types of impairments, such as limitations in vision, hearing, cognition, movement and speech. Accessibility testing is an important part of the process to achieve universally designed solutions (1).

Software development is a complex process where many people are involved. During the last 20 years, the profession has improved the quality and complexity to a level where it is not easy for a single person to have complete overview over all functionality. To distribute the burden of fixing bugs and avoid regression, the industry have introduced techniques that were uncommon 20 years ago; unit testing, faster and smaller iterations, automated building of source code, automated execution of integration tests, and so on. However, accessibility testing is still mostly performed by humans at the end of software cycles (2). Even though late accessibility testing is costly (3), projects seem to ignore this fact in the heat of the moment. Research also suggest that developers need easy methods of implementing accessibility in order to increase the overall accessibility of digital solutions (4; 5).

*All authors were with the Department of Applied Research, Norwegian Computing Center, Oslo, Norway.

This paper was presented at the NIK-2019 conference; see http://www.nik.no/.

To reduce the costs of accessibility testing, it seems wise to follow an automated approach which is well integrated in the software development process. Furthermore, it has been argued that automated accessibility testing avoids repetitive and manual testing, which allows teams to focus on delivering flawless and highly usable software (6).

In this paper, we try to answer the research question in how far automatic accessibility checkers can in fact add value to today's continuous-deployment projects. Doing so, we focus on testing tools for web development, since this is the primary medium for most software projects today. The main contributions of our work are the overview and assessment of available automatic tools, and the analysis of the underlying rulesets.

The remainder of the paper is organized as follows: After the discussion of related work in Section 2, we give an overview over the most common automated accessibility testing tools in Section 3. Our methodology is presented in Section 4. We present results from the evaluation of tools in Section 5, before different strategies for integration in build processes are discussed in Section 6. Finally, we summarize and highlight future research directions in Section 7.

2 RELATED WORK

Multiple studies have evaluated automated accessibility tools (7; 8; 9). The tools find different accessibility issues, but there is no single tool to find all the issues (7; 9). Even when results from the tools are combined, almost 1/3 of the issues are not found (10). Often, these tools must be invoked manually. Some of them are prone to poor usability and lack the adequate functionality for web developers (11; 9). Therefore, web pages should be tested for accessibility in at least three different ways: automated testing for code compliance and compatibility with assistive technology, manual testing by experts, and user testing involving persons with disabilities (12).

It is very cost-efficient to automatically uncover (and fix) accessibility problems as early as possible in the development process (3). In that way, testing that involves humans can concentrate on issues that cannot be checked automatically. Therefore, although the use of automated accessibility tools has limitations, using such tools can be an important step towards achieving more accessible solutions. Also, in order to ensure and maintain the achieved accessibility level, one is dependent on routine testing and the use of automated evaluation tools (13; 14).

Most studies focus on automated tools that must be operated by a user, either in the browser or using a native application. Some researchers discuss various accessibility testing tools to be integrated into a build process (15; 16), but the tools are not compared with each other, and only with other accessibility evaluation methods.

3 ACCESSIBILITY TESTING TOOLS

There is no global overview over all the possible tools for accessibility testing, and in particular not for tools that can be integrated in a build process. The most complete overview over available testing tools for web is W3C's *Web Accessibility Evaluation Tools List* (17). However, the list is slightly outdated and also missing popular alternatives, such as *Automated Accessibility Testing Tool* (AATT) from Paypal (18). Therefore, we conducted an extensive online search and seeks on software

project sites like Github and Bitbucket to identify accessibility testing tools possible to integrate in a build process.

According to the W3C, most existing tools cannot be run from the command line (17), and very few provide an API that enables their integration into a build process. Only tools and libraries that were possible to integrate fully into an automatic build process for web development were considered, and thus many popular accessibility checkers (like SiteImprove) were disregarded in our study.

We identified 11 tools that can be integrated in a build process for web development, along with the ruleset the tools is based on, enlisted in Table 1. A ruleset is the collection of a limited number of tests to carry out.

Table 1: Overview of automatic testing tools. A grey background marks active development

#	*Tool*	*Ruleset*	*Active*	*Envir.*
1	aXe	aXe-core	yes	Node
2	pa11y	HTML-CS	yes	Node
3	GADT	self-defined	no	Node
4	AATT	aXe-core, HTML-CS	yes	Node
5	accessibilityjs	accessibility.js	no	Node
6	The A11y Machine	HTML-CS	no	Node
7	Access Continuum	self-defined	yes	Java
8	Asqatasun-Runner	self-defined	yes	Java
10	Webhint	aXe-core	yes	Node
11	Google Lighthouse	aXe-core	yes	Node

Some tools (like tenon.io and WAVE's stand-alone API) are possible to integrate in a build process, but they cost money. They are accessible as a web service which hurts performance by introducing a network roundtrip for every test. This can be significant even for a small project if the code is built and tested often. Service tools were therefore not include in the evaluation.

To compare the different tools, we identified 32 criteria we deemed relevant from a development and integration perspective. We used already established criteria (19; 9; 20) before trimming the list down to 23 criteria that was essential for evaluating all the different tools.

We also removed redundant criteria because they were already covered by the ruleset. Criteria that covered setup complexity and very technical elements like DOM coverage were ignored. We will not discuss the 23 criteria here, but focus on a smaller subset that were important when selecting the correct tool.

Three criteria proved to be more significant when choosing tools for further investigation: ruleset, active development and environment. It is critical with an active development to make sure that bugs are fixed, new rules are added, and that the documentation is maintained. It is also important that a Node (JavaScript) environment is supported, since this is the standard environment for modern web development and build processes (21; 22).

If we eliminate all tools that do not have an active development or Node as environment, we can reduce the list to five tools, number 1, 2, 4, 10 and 11, marked with a grey background in Table 1. However, we can further trim the list down to the different rulesets deployed, since there are only two major rulesets: aXe-core (23) and HTML-CodeSniffer (HTML-CS) (24). They are designed to test not only

the accessibility of entire websites and pages, but also markup snippets and other HTML-based interfaces. Both are available on Github. This choice boils our 11 automatic tools down to essentially two types; those that use the aXe-core or the HTML-CS ruleset. This finding is confirmed by an extensive search which returned one or both of these tools in almost all search results.

Until more popular rulesets emerge, it is probably a wise choice to select a testing tool that is based on either aXe-core or HTML-CS. We have also seen that existing tools that are not based on one of these rulesets (but rather use their own) are starting to adopt them. E.g., Google turned down development of their GADT tools based on a self-defined ruleset and created the Lighthouse tool instead which is based on aXe-core.

Even though we focus on the rulesets used by the tools in this paper, there are of course many differences between the actual tools. Which specific tools that is the best fit for a project depends on many factors, and we suggest to use the Appendix A as a guidelines for deciding on a particular tool. However, as we will discuss in the next sections, it's important to select the best ruleset first.

4 METHODOLOGY

To evaluate the rulesets, they were tested against a site with defined accessibility problems. We used the W3C *before and after demonstration* site (25). This site has four pages in an inaccessible and an accessible version, with errors defined by persons within the Web Accessibility Initiative (26). This makes it possible to check how many issues a ruleset is able to discover in the inaccessible version, and how many false positives a ruleset reports in the accessible version.

We also checked which WCAG 2.1 guidelines (27) the rulesets claim to detect and which WCAG Level (A/AA/AAA) they apply to, and the results are shown in Table 2. We used the latest version of both rulesets (versions as listed in the table), and we used the aXe tool (28) when checking the aXe-core ruleset and the pa11y tool (29) when testing the HTML-CS ruleset. Both rulesets have multiple rules for a single WCAG guideline, and they appear to be quite similar in terms of number of rules; aXe-core has 56 rules in total while HTML-CS has 61 rules in total.

Both rulesets distinguish between WCAG levels. Furthermore, HTML-CS operates with levels of found issues; issues classified as errors are detectable by the tool alone, issues that are classified as notices and warnings requires manual inspection. Only rules that classify as errors were considered in this study.

The aXe-core ruleset does not contain any rules that requires manual inspection, but rules that are classified as "best practice" were excluded since they are not a part of the WCAG specification.

5 RESULTS

Table 2 gives an overview of the rulesets and their WCAG 2.1 coverage. A success criteria was defined as "covered" if one or more rules was present in the ruleset. This does not mean that a tool checks for every possible violation of the criteria, but that the tool has checks for at least one violation of the criteria.

In WCAG 2.1 (27), there are 30 guidelines for level A, 20 guidelines for level AA, and 28 guidelines for level AAA; a total of 78. Thus, the best ruleset covers only 31% (32% in WCAG 2.0) of the WCAG criteria, as indicated in Table 2. On the

Table 2: Overview of ruleset and their WCAG 2.1 coverage

Tool	*Level A*	*Level AA*	*Level AAA*	*Total*
aXe-core (3.2.2)	16 (53%)	6(30%)	2(7%)	31%
HTML-CS (2.2.0)	10(33%)	2(10%)	4(14%)	21%

positive side, the best ruleset covers 53% of the level A criteria, but this illustrates that even the best ruleset / automatic tool has a lot of potential for improvement. This is as expected and also reported by other studies (7).

In our evaluation we focused on three key parameters: True Positives (TP), False Positives (FP), and False Negatives (FN). TP is where a ruleset flags an issue, which really is an issue, in the inaccessible version. FP is where a ruleset reports a WCAG error that is not defined as an issue in the accessible version. FN is when a ruleset does not find a defined issue in the inaccessible version. We also conducted an expert evaluation if a ruleset reports a FP, since it potentially could be an error in the accessible version.

Based on these parameters, we can calculate a tool's precision (or correctness), and recall (or completeness) according to Eq. 1 and Eq. 2. Both these metrics are important to analyze since precision tells us how many of the reported issues are actually correct, and recall quantifies the degree of correctly identified issues (30). The range for both precision and recall is from 0 to 1, where 0 is the worst result and 1 is the best result. The number can be multiplied with 100 to give a percentage.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

The results from the aXe-core ruleset are found in Table 3. Here, the ruleset is evaluated against the inaccessible version where the W3C has deliberately planted 110 issues that violate WCAG guidelines. The true number of errors is higher, as many guidelines are violated by multiple errors. We have counted the violations of WCAG guidelines and not the total number of errors.

As Table 3 shows, the aXe-core ruleset does not find many of the planted WCAG violations, with a high number of FN and a low recall for all pages. However, aXe-core does not report any FP and achieves a perfect precision once it detects a violation.

Table 3: Ruleset results for aXe-core from the inaccessible version

Page	*TP*	*FP*	*FN*	*Precision*	*Recall*
Home	8	0	19	1.0	0.30
News	7	0	20	1.0	0.26
Ticket	8	0	20	1.0	0.29
Survey	7	0	21	1.0	0.25

For the HTML-CS ruleset, the results from the inaccessible version are found in Table 4. The same pattern is found here, with a low recall and perfect precision. However, the HTML-CS ruleset has a worse recall than aXe-core.

Table 4: Ruleset results for HTML-CS from the inaccessible version

Page	*TP*	*FP*	*FN*	*Precision*	*Recall*
Home	5	0	22	1.0	0.19
News	5	0	22	1.0	0.19
Ticket	5	0	23	1.0	0.18
Survey	5	0	23	1.0	0.18

The aXe-core ruleset finds almost all WCAG guideline violations that HTML-CS finds, but HTML-CS does not find the following WCAG violations: 2.4.1 (Bypass Blocks), which is level A, 2.4.4 (Link Purpose), which is level A, and 3.3.2 (Labels or Instructions), which is level A. Since all are level A, it is a little surprising that the HTML-CS ruleset does not find them, considering that they should be programmatically easy to detect.

Both rulesets finds these WCAG guideline violations: 1.1.1 (Non-text Content), which is level A, 1.3.1 (Info and Relationships), which is level A, 1.4.3 Contrast (Minimum), which is level AA, 3.1.1 (Language of Page), which is level A, and 4.1.2 (Name, Role, Value), which is level A. It is not very surprising, though, that both rulesets find violations for the mentioned WCAG guidelines as these have well defined rules.

The results from the accessible version are shown in Table 5. According to the W3C demo site (25), there should not be a single guideline violation, and we have only listed FP in the table. Only the HTML-CS ruleset reports a WCAG violation.

Table 5: Number of reported FP in the accessible version

Page	*aXe-Core FP*	*HTML-CS FP*
Home	0	0
News	0	0
Ticket	0	1
Survey	0	1

It should be mentioned that the false positive reported by HTML-CS is a difficult case. It is claimed to be a violation of WCAG guideline 1.3.1, and HTML-CS reported the violation with the description *Incorrect headers attribute on this td element*. This is, however, incorrect as the td element uses the correct header references (31), even though one of the labels is not strictly above the column. This illustrates how difficult it is to interpret the guidelines.

Based on the total number (sums) of TP, FP, and FN, we can calculate the total precision and recall as shown in Table 6. We also calculate the $F_1 score$ according to Equation 3. This metric is a harmonic average of both precision and recall (30) and makes it easier to compare the rulesets against each other. The $F_1 score$ reaches its best value at 1 (perfect precision and recall) and worst at 0.

$$F_1 score = 2 \times \frac{precision \times recall}{precision + recall} \tag{3}$$

As Table 6 show aXe-core has a decent $F_1 score$ with 0.43, while HTML-CS is slightly worse with 0.30. Even though aXe-core has a significant higher $F_1 score$ than HTML-CS, both rulesets achieves reasonable results considering they can only check for programmatic violations.

Table 6: Total results from all pages and both versions

Ruleset	*TP*	*FP*	*FN*	*Precision*	*Recall*	*F1 score*
aXe-core	30	0	80	1.00	0.272	0.429
HTML-CS	20	2	90	0.909	0.181	0.303

The aXe-core tool both finds more violations (TP) and is more reliable (no FP) as compared to HTML-CS. So, given the version 3.2.2 of aXe-core and version 2.2.0 of HTML-CS, aXe-core is clearly the better option. However, as both tools are under active development, our tests should be repeated from time to time to mirror the improvements of the development team's latest patches.

6 DISCUSSION

In our evaluation, both rulesets achieved low recall scores (0.43 and 0.30), which is not unexpected since automated accessibility checkers are far from enough to find all accessibility issues on a webpage. They struggle in particular to find violations of WCAG guidelines that requires deduction about context and layout (32).

However, the high precision scores (1.00 and 0.91) show that the rulesets can be trusted in what they report. This is reassuring and indicates that they are safe to integrate in an automated build process. A high number of FPs causes unnecessary noise during the build process where humans manually have to inspect if reported violations are actual errors or not.

Unless a test breaks, code is not compilable, or something unexpected happens, most software teams want the build process to operate in the background. This strategy is actually imperative in continuous deployment where several hundred or even thousands of builds happens every day (33; 34).

It is not obvious what the best integration approach for accessibility testing tools is. For example, should an accessibility violation break the whole build similar to a unit test failure? Or should only a violation of WCAG Level A guidelines interrupt deployment? Should particular violations be tolerated in prototyping, and/or in development, or even production?

We know from other studies that accessibility testing is often considered a burden and therefore not prioritized (35). This means that a possible consequence of a noisy accessibility testing tool is that teams might decide not to integrate accessibility tools altogether. However, it is not advisable to go down that road, as not having accessibility testing as part of an automated build process is considered to be an anti-pattern (36), and accessibility testing should therefore be fully integrated in an automated build process for cost effectiveness.

An argument against this, however, is that automated accessibility testing does not find all accessibility issues. While this is true to some extent, it is not a good reason not to take control of the issues that they do find. A high degree of automation will also likely ease the burden of manual accessibility testing by removing accessibility issues that are easy to find programmatically.

Another argument against integrating accessibility testing tools in the build process is that it takes time to fix the accessibility violations. This concern is debunked by the benefits of providing accessible software (37; 38), and because more and more countries are introducing accessibility laws (39; 40).

To change the current software testing approach and mindset, accessibility

testing tools thus need to be integrated fully into modern build processes. The above arguments also advocate for following the approach that, if an accessibility violation is found, this event should break the build in the same manner as unit tests. It will cause a little overhead in the beginning for teams that are not used to work with accessibility, but once the team members understand why the build is failing, they are anticipated to start focusing on how to avoid introducing those bugs.

Having accessibility testing tools as part of the build process will then also provide a fail-safe for web development teams. A common misconception is that once a webpage has been made accessible, the work required to keep it accessible is done. This is grossly incorrect since a webpage is in continuous development as teams change, features are altered or removed, new features are added, and since the software is constantly evolving. This means that a webpage that once was accessible can suddenly become inaccessible if no attention is paid to accessibility.

7 CONCLUSION

Based on an overview of the most common automatic accessibility tools for software build integration, we discovered that the most significant factor was the ruleset. There were two primary rulesets among all the active tools: aXe-core (23) and HTML-CS (24). Our investigation revealed moderate differences in quality, and aXe-core outperformed HTML-CS particularly in terms of $F_1 score$. However, both tools still have considerable potential for improvement when it comes to WCAG coverage and recall. This could be remedied by adding more tests to the rulesets, by converting tests from manual to automatical execution, and by improving the quality of existing tests.

We argue that the result presented here is independent of tools we might have missed, as such tools are likely to be based on either aXe-core or HTML-CS, and we have based our study on those rulesets and not the actual testing tool. It is, however, nevertheless important to ensure that the selected testing tool uses the latest version of the rulesets to mirror the latest code improvements.

Underlying rulesets and automatic tools are critical enablers for having accessibility testing fully integrated in an automated build process. They are also a prerequisite if teams want accessibility violations to interrupt the deploy process. Their successful integration is thus a promising strategy to achieve a continuous focus on accessibility in software development.

We conclude that some automatic accessibility checkers in fact have the potential to add value to today's continuous-deployment projects if integrated properly. Added value also implies, though, that the right strategy for handling accessibility errors is chosen. The tools' two underlying rulesets have different quality, and our analysis shows that the use of aXe-core is currently preferable over HTML-CS.

Two final comments. First, we have focused in this work on accessibility testing tools for web development because the area has various tools available to check accessibility issues, and because web is the dominating medium in the industry. Our method can, however, be extended to typical desktop and native software development, assuming the sufficient ruleset and tools are developed. Second, to our knowledge, there is no resource other than W3C's *before and after demonstration* site that has a quality-assured and controlled accessible and inaccessible version of a set of webpages. We urge the accessibility community to create a new and modern

webpage resource that can benefit the benchmarking of accessibility testing tools in the future. Such a resource should use typical scenarios like shopping, finding content and other complex interactions which are commonly used. In addition, an upgrade to WCAG 2.1 is needed, since the W3C's *before and after demonstration* site only covers WCAG 2.0.

8 ACKNOWLEDGMENT

This work has been supported by the ITIK project funded by the Norwegian Directorate for Children, Youth and Family Affairs in the UnIKT program, grant number 62127.

A RULESET CRITERIA

Table A contains all the 22 criteria that were used in evaluating the different accessibility testing tools. The most significant criteria are marked with a grey background.

Table 7: Overview of criteria

#	*Criteria*	*Description*
1	Automatic testing	Is it possible to run tests automatically from a script?
2	Build integration	Can be integrated in the development/build process
3	Local tests	Can be installed and run locally without external API calls
4	Mature	Is the tool considered stable or in beta?
5	License	License type and name
6	Active development	Is the tool under active development?
7	Community activity	Is the community active?
8	Documentation quality	Is documentation available? What is the quality of the documentation?
9	Flexible integration	Can the tool be integrated with more than one framework or technology?
10	Requirements	Which frameworks and technologies are required?
11	Ruleset	Which rulset(s) are used?
12	Run configuration	Is it possible to include/exclude rules?
13	Output severity options	Is it possible to include/exclude results with varying degree of severity?
14	Output formating options	Is it possible to specify different output formats?
15	Login capabilties	Is it possible to test pages that require login?
16	Complicated setup	How much work is required to setup the tool?
17	Performance	How long does it take to perform a test on a webpage with average complexity?
18	Supports testing of frames	Is it possible ot perform test of frames and it's content?
19	Disrupts process	Will an error in the test stop the process it is run within?
20	Expandable rule set	Is it possible to code and add new rules?
21	CSS coverage	Does the tool test for possible errors in css files?
22	Code inspection	Does the tool provide indication of where issues are located in the source code?
23	Environment	What environment is supported?

References

[1] K. S. Fuglerud, *Inclusive design of ICT: The challenge of diversity.* Dissertation for the Degree of PhD, University of Oslo, Faculty of Humanitites, 2014.

[2] D. Swallow, C. Power, H. Petrie, A. Bramwell-Dicks, L. Buykx, C. A. Velasco, A. Parr, and J. O. Connor, "Speaking the language of web developers: Evaluation of a web accessibility information resource (WebAIR)," in *Lecture Notes in Computer Science*, vol. 8547 LNCS, 2014.

[3] S. Horton and D. Sloan, "Accessibility in Practice: A Process-Driven Approach to Accessibility," in *Inclusive Designing.* Cham: Springer International Publishing, 2014, pp. 105–115.

[4] M. Baez and F. Casati, "Agile development for vulnerable populations: Lessons learned and recommendations," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Society.* ACM, 2018, pp. 33–36.

[5] M. Crabb, M. J. Heron, R. Jones, M. Armstrong, H. Reid, and A. Wilson, "Developing accessible services: understanding current knowledge and areas for future support." 2019.

[6] C. Putnam, M. Dahman, E. Rose, J. Cheng, and G. Bradford, "Best practices for teaching accessibility in university classrooms: cultivating awareness, understanding, and appreciation for diverse users," *ACM Transactions on Accessible Computing (TACCESS)*, vol. 8, no. 4, p. 13, 2016.

[7] T. Halbach and W. Lyszkiewicz, "Accessibility checkers for the web: How reliable are they, actually?" in *Proceedings of the 14th International Conference WWW/Internet*, vol. 2015, 2015, pp. 3–10.

[8] G. Brajnik, "Comparing accessibility evaluation tools: a method for tool effectiveness," *Universal Access in the Information Society*, vol. 3, no. 3-4, pp. 252–263, 2004.

[9] S. Trewin, B. Cragun, C. Swart, J. Brezin, and J. Richards, "Accessibility challenges and tool features: an ibm web developer perspective," in *Proceedings of the 2010 international cross disciplinary conference on web accessibility (W4A)*. ACM, 2010, p. 32.

[10] Gov.uk. How do automated accessibility checkers compare? [Online]. Available: https://alphagov.github.io/accessibility-tool-audit/

[11] H. Petrie, N. King, C. Velasco, H. Gappa, and G. Nordbrock, "The usability of accessibility evaluation tools," in *International Conference on Universal Access in Human-Computer Interaction*. Springer, 2007, pp. 124–132.

[12] C. L. Vera, in *The 2018 ICT Accessibility Testing Symposium: Mobile Testing, 508 Revision, and Beyond*, Arlington, VA, USA.

[13] M.-L. Leitner, C. Strauss, and C. Stummer, "Web accessibility implementation in private sector organizations: motivations and business impact," *Universal Access in the Information Society*, vol. 15, pp. 249–260, 2016.

[14] M. L. Sánchez-Gordón and L. Moreno, "Toward an integration of web accessibility into testing processes," in *Procedia Computer Science*, vol. 27, 2013.

[15] A. Bai, K. Fuglerud, R. Skjerve, and T. Halbach, "Categorization and comparison of accessibility testing methods for software development." *Studies in health technology and informatics*, vol. 256, pp. 821–831, 2018.

[16] H. L. Antonelli, L. Sensiate, W. M. Watanabe, and R. P. de Mattos Fortes, "Challenges of automatically evaluating rich internet applications accessibility," in *Proceedings of the 37th ACM International Conference on the Design of Communication*. ACM, 2019, p. 32.

[17] Web accessibility evaluation tools list. [Online]. Available: https://www.w3.org/WAI/ER/tools/

[18] Automated accessibility testing tool (aatt). [Online]. Available: https://github.com/paypal/AATT

[19] Selecting web accessibility evaluation tools. [Online]. Available: https://www.w3.org/WAI/test-evaluate/tools/selecting/

[20] G. Brajnik, "Automatic web usability evaluation: what needs to be done," in *Proc. Human Factors and the Web, 6th Conference*, 2000.

[21] D. Herron, *Node. js web development: server-side development with Node 10 made easy*. Packt Publishing Ltd, 2018.

[22] Measuring web framework popularity so you can find interesting frameworks to check out. [Online]. Available: https://hotframeworks.com/

[23] Deque. axe-core ruleset. [Online]. Available: https://github.com/dequelabs/axe-core

[24] Squiz. Html codesniffer. [Online]. Available: https://squizlabs.github.io/HTML_CodeSniffer/

[25] Before and after demonstration. [Online]. Available: https://www.w3.org/WAI/demos/bad/Overview.html

[26] W3C. Web accessibility initiative. [Online]. Available: https://www.w3.org/WAI/

[27] ——. Wcag 2.1. [Online]. Available: https://www.w3.org/TR/WCAG21/

[28] Deque. Axe-core. [Online]. Available: https://axe-core.org/

[29] T. Pa11y. pa11y. [Online]. Available: http://pa11y.org/

[30] D. L. Olson and D. Delen, *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[31] W3C. Using id and headers attributes to associate data cells with header cells in data tables. [Online]. Available: https://www.w3.org/TR/WCAG20-TECHS/H43.html

[32] M. Tollefsen and T. Ausland, "A practitioner's approach to using wcag evaluation tools," in *2017 6th International Conference on Information and Communication Technology and Accessibility (ICTA)*. IEEE, 2017, pp. 1–5.

[33] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[34] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 21–30.

[35] T. Halbach and K. S. Fuglerud, "Reflections on cost-benefit analyses concerning universal design of ICT solutions," in *Proceedings of 10th International Conference on Interfaces and Human Computer Interaction*. IADIS, 2016.

[36] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of systems and software*, vol. 138, pp. 52–81, 2018.

[37] W. T. Andrade, R. G. d. Branco, M. I. Cagnin, and D. M. B. Paiva, "Incorporating accessibility elements to the software engineering process," *Advances in Human-Computer Interaction*, vol. 2018, 2018.

[38] S. Schmutz, A. Sonderegger, and J. Sauer, "Implementing Recommendations from Web Accessibility Guidelines: A Comparative Study of Nondisabled Users and Users with Visual Impairments," *Human Factors*, vol. 59, no. 6, pp. 956–972, 2017.

[39] C. M. Law, P. D'Intino, J. Romanowski, and R. Sinclair, "Priorities for the field: Management and Implementation of Testing within Accessibility Programs," in *The 2018 ICT Accessibility Testing Symposium: Mobile Testing, 508 Revision, and Beyond*. Arlington, VA, USA: Accessibility Track Consulting, LLC, 2018, pp. 141–153.

[40] W3C. Web accessibility laws policies. [Online]. Available: http://www.w3.org/WAI/policies/