Paper Title\* (use style: paper title)

Uday Jain  
Department of Software Engineering  
Blekinge Institute of TechnologyKarlskrona, Sweden  
uday9k@gmail.com line 1: 2nd Given Name Surname  
line 2: *dept. name of organization*   
line 3: *name of organization*line 4: City, Country  
line 5: email address or ORCID

*Abstract*—Pending.

Keywords—component, formatting, style, styling, insert (key words)

# Introduction (*Heading 1*)

Pending

# Related Work

Modern day software development cycles often include small iterative cycles of continuous development and testing, efficiently integrating LEAN software testing principals [1] [2]. Recent literature also suggests the claimed benefits of test driven development (TDD) may not be due to its distinctive test-first dynamic, but rather because TDD like processes encourage fine grained, steady steps that improve focus and flow [3] [4]. Enough evidence is now available to understand efficiency of testing systems given goals and constraints [5], establishing the precedence to automation [6].

With realisation of testing as apart of software development cycle and integration into industrial process, technology stacks have evolved [7] [8] to integrate continuous integration and continuous delivery (CICD) with testing as a service [9] [10], pushing the limits on time to delivery. Rapid development in open-sourced tools in recent years has made it easier to combine multiple components offered by the community and stich together a comprehensive automated testing environment customised to individual use cases [11]. Integration techniques which were development intensive [12] have now become available to use out of the box [13]. This rapid development has paved the way for automated testing [14] [15] its further optimizations [16] [17] [18].

As a part of this paper, we have used mutation testing along with code coverage reports to evaluate the strengths of our testing suit. Multiple studies have indicated use of coverage metrics and its variants can be used as an indicator of test suite quality [19] [20]. With high popularity among open source community, mutation testing has also proved to play a significant role in identifying defects and help improve test suit quality [21] [22]. Pitest & OpenClover have been identified as a popular open source Jenkins plugin that provide easy to setup mutation testing and code coverage analysis for maven project [23] [24].

# Methodology

To build a customized environment automating testing of software updates we started with selection of KPI (key performance indicators) that we want to test the performance with each of the commits to GitHub repository. As a part of this exercise, we started with understanding the process of development and different types of tests written. We then evaluated inputs from developers and testers along with product managers (PM) to understand the requirements based on KPI that would provide most meaningful inputs for further development, iterations and evaluating the build and its history. To optimize the metrics, we only considered the ones that provide nonoverlapping information and ones that were identified by more than 3 developers in a squad of 5 developers. The goal is to provide simple reports that cover most use cases across the teams.

Leveraging the small size of the team, we conducted interviews and brainstorming sessions to identify the following list of KPIs:

* %Code Coverage
* %Test passed
* % Mutation Coverage
* % Test Strength

Based on the identified KPIs we then outlined most effective tools that generate such reports. Effectiveness was estimated based on following parameters:

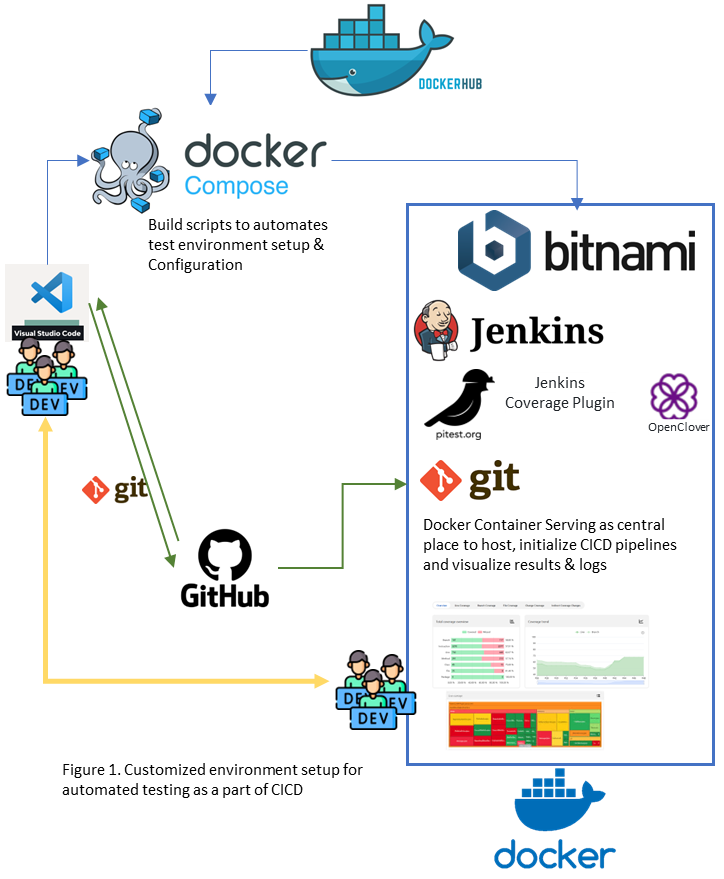
* Open-Source availability
* Community support and active contributors
* Recency in Commits
* Issues identified
* Ease of implementation
* Level of detail

To identify an appropriate technology stack, we outline following parameters:

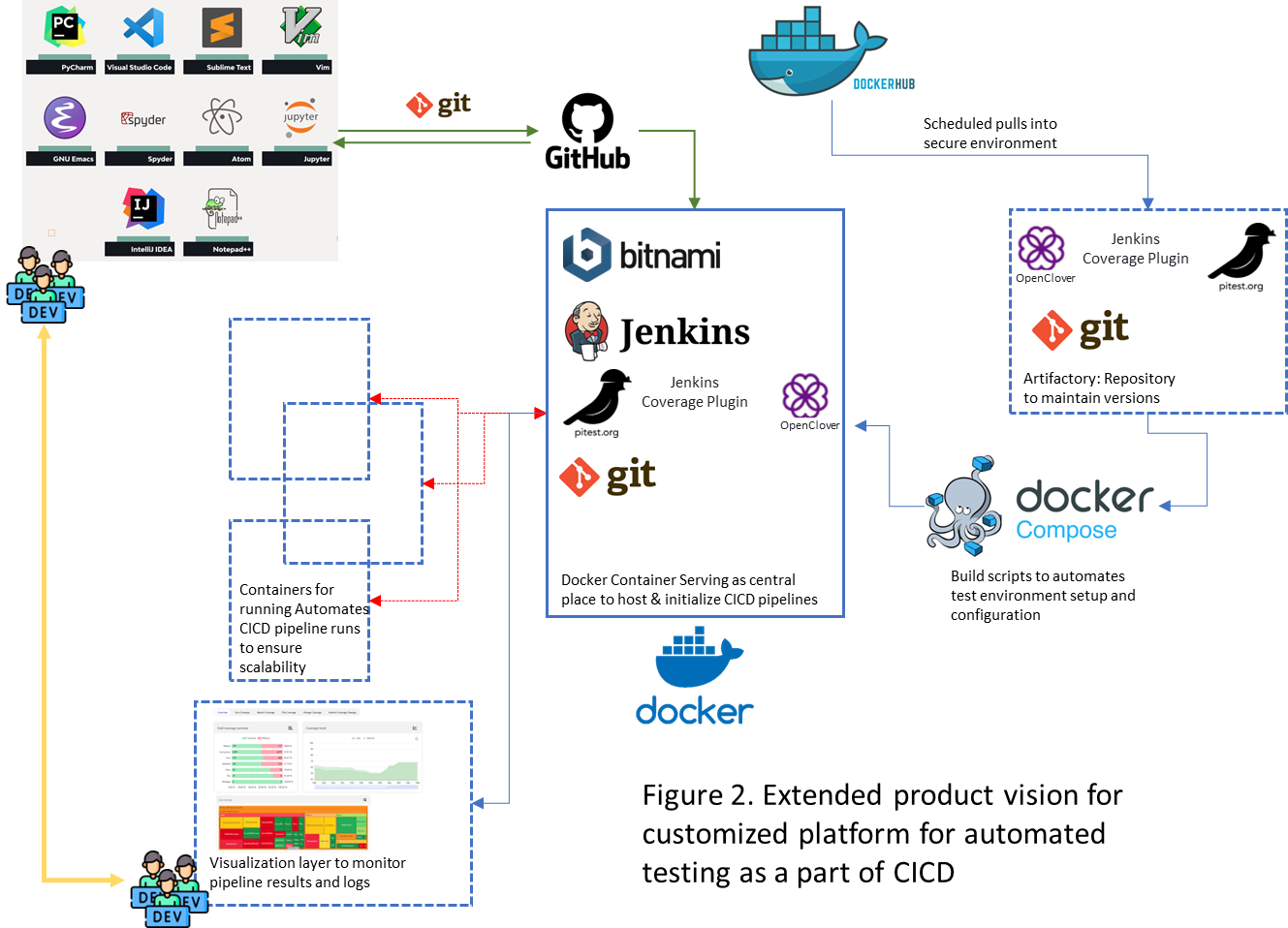
* Scalable: both horizonal and vertical scalability in terms of computational and storage resources
* Ease of customization
* Compatibility with available tools and resources
* Ease of implementation and iterative development
* Popularity among developers and testers

Since GitHub was the preferred choice to store code base, among the contenders, a docker based Jenkins container that connects to GitHub was found most suitable. Additionally, GitHub can easily connect to any IDE (Integrated Development Environment) that the developers choose to use, a platform for automated testing using the technology stack required little changes to the existing ways of working within the team.

Bitnami offers open-sources Jenkins based docker images that can be customized to suit the needs of the organization. To choose additional plugins based on our identified KPIs, prioritization using parameters listed above was done. We started development using Pitest, OpenClover and Test Result Analyzer. These plugins efficiently capture the all required metrics and provide a detailed report that can be further customized to individual needs. Figure 1 captures the environmental setup of the automation testing platform MVP (minimum viable product) that we used as a part of the exercise.



The platform (Figure 1) provides extensive flexibility to develop additional customizations and modularizations to ensure stability and security among the systems. Figure 2 outlines the platform vision with its components to be developed as the need and use grows over time.



All tests runs were performed on a single laptop with 32GB RAM (16GB available to container running) and 8 processor (4 core available to container running) to ensure comparison among results. To minimize computation time with each build used “DwithHistory” as a default option when running Pitest. To test the environment we used open-source project Joda-Time committed to GitHub repository [here](https://github.com/SpikeStriker/lecture_ci.git).

# Results

##### To build the environment using a docker compose file it took a total of 25 minutes once the images were downloaded. However, to identify right set of combination for package and plugin versions and respective configurations, it took more than 30 iterations.

##### Building the project and running all tests (4212 tests in total, 100% passed) took 9 minutes and 14 seconds when complied to publish OpenClover reports. The build running PIT mutations to publish both OpenClover reports and PIT mutation reports however took 1 hour and 35 minutes of which mutation analysis took 1 hour and 30 minutes. During the run 9969 PIT mutation were generated and 7812 or 78% mutations were killed using existing tests. Among them 1018 mutation had no coverage thereby providing a test strength of 87%. In total 118,302 tests were run (approximating 11.87 tests per mutation) and a line coverage of 12954/14363 (90%) was achieved for mutated classes.

During exploration 2 sets of tests cases were used:

* With all test (*org.joda.time.TestAll.suite, org.joda.time.chrono.TestAll.suite, org.joda.time.chrono.gj.TestAll.suite, org.joda.time.convert.TestAll.suite, org.joda.time.field.TestAll.suite, org.joda.time.format.TestAll.suite, org.joda.time.tz.TestAll.suite*)
* With only *org.joda.time.TestAll.suite* test.

Changing the number of test suits had little impact on the total run time.

Figures 3 & 4 capture the results of Pit Test while Figure 5-8 capture OpenClover results for built with all test cases and Figure 9-12 capture OpenClover results for built with only *org.joda.time.TestAll.suite* test cases.

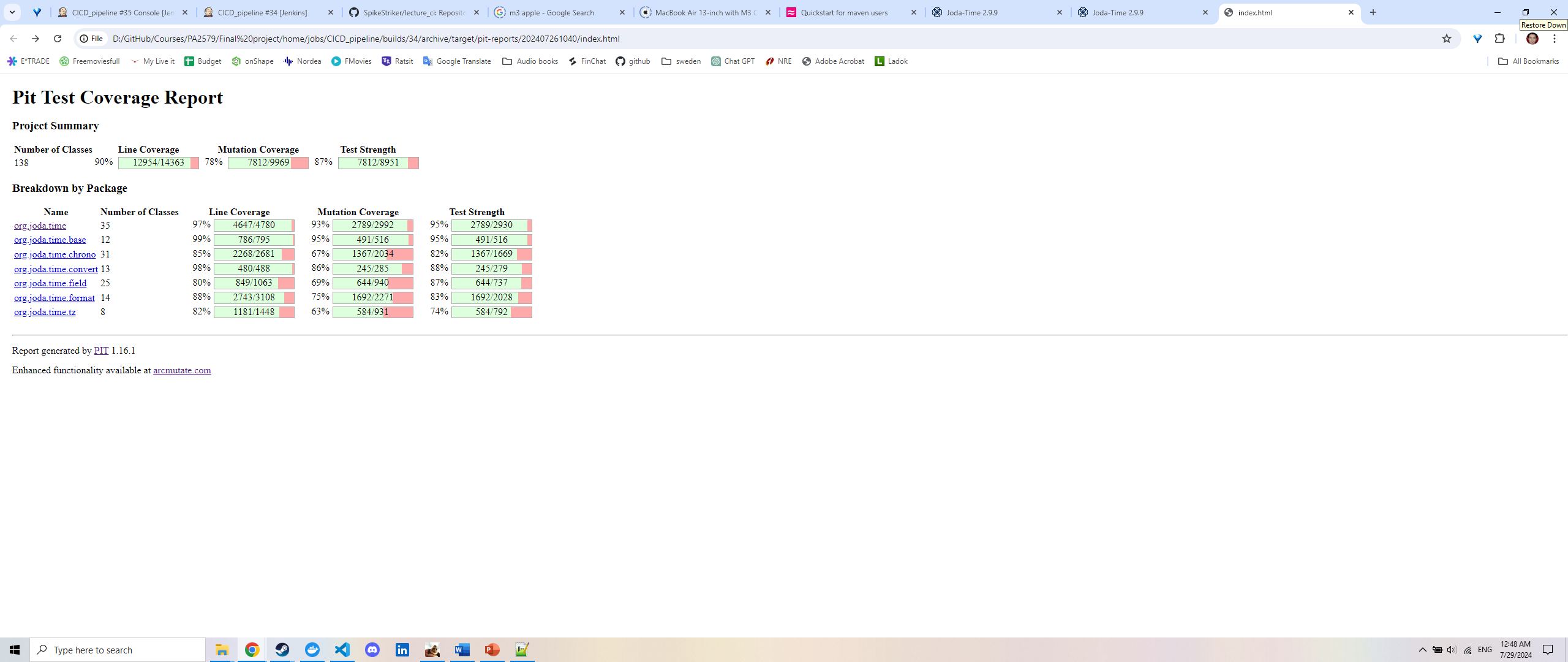


Figure 3. PIT Mutation analysis report breakdown by Package

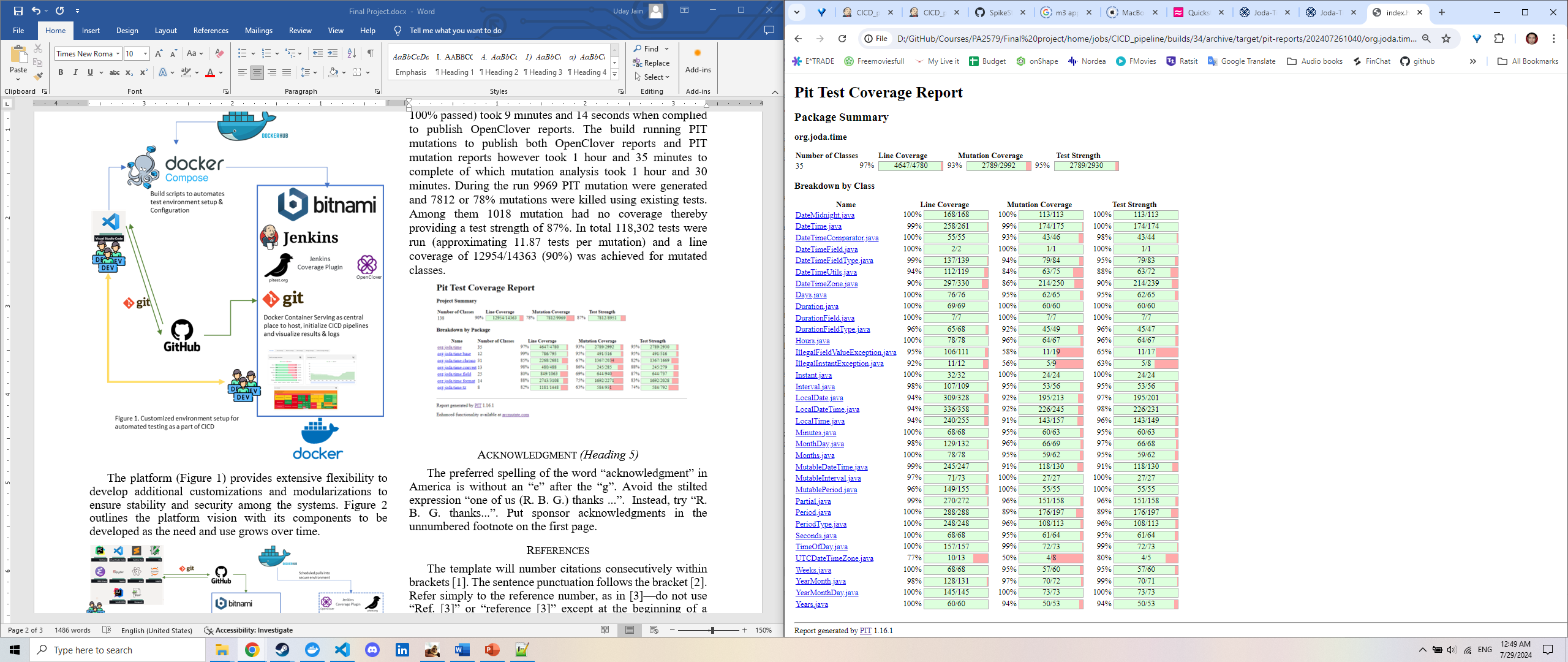


Figure 4. PIT Mutation analysis report breakdown by Class eg. org.joda.time

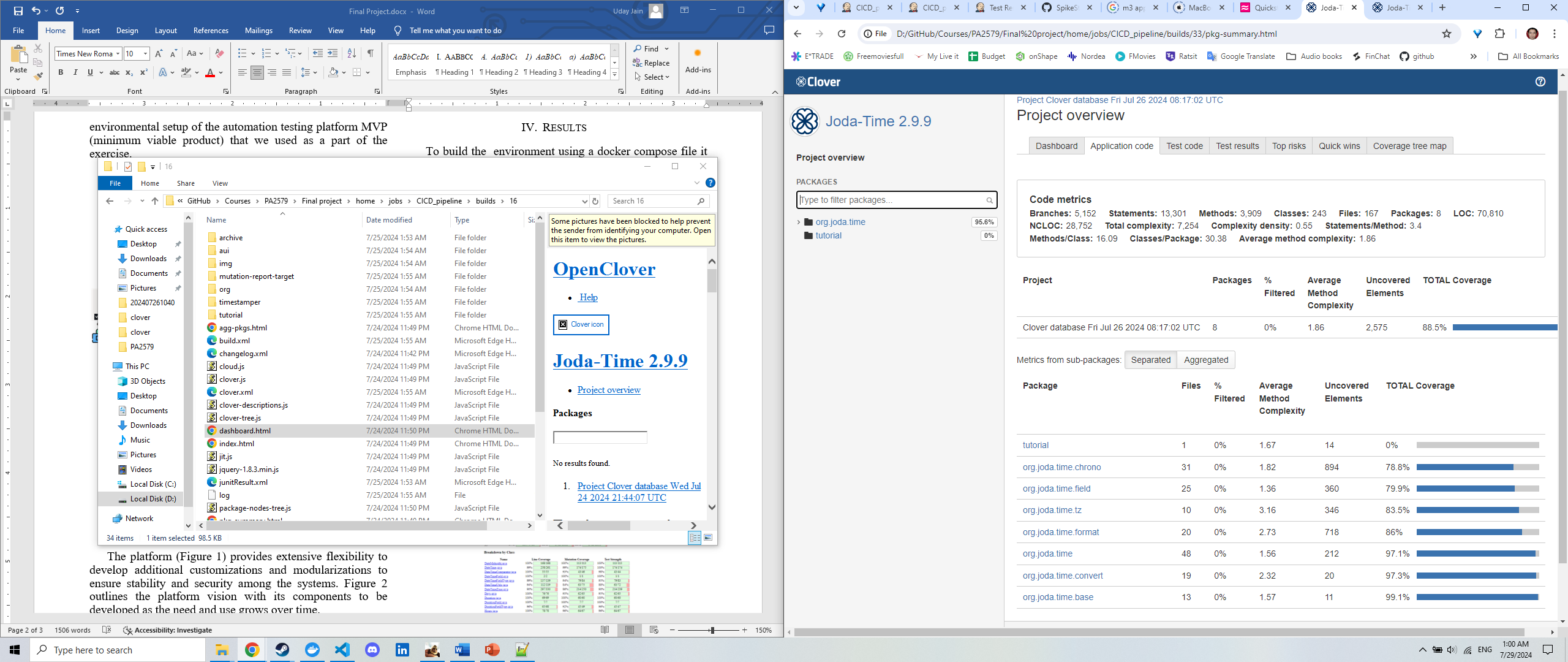


Figure 5. OpenClover report for application code coverage analysis overview when running all tests

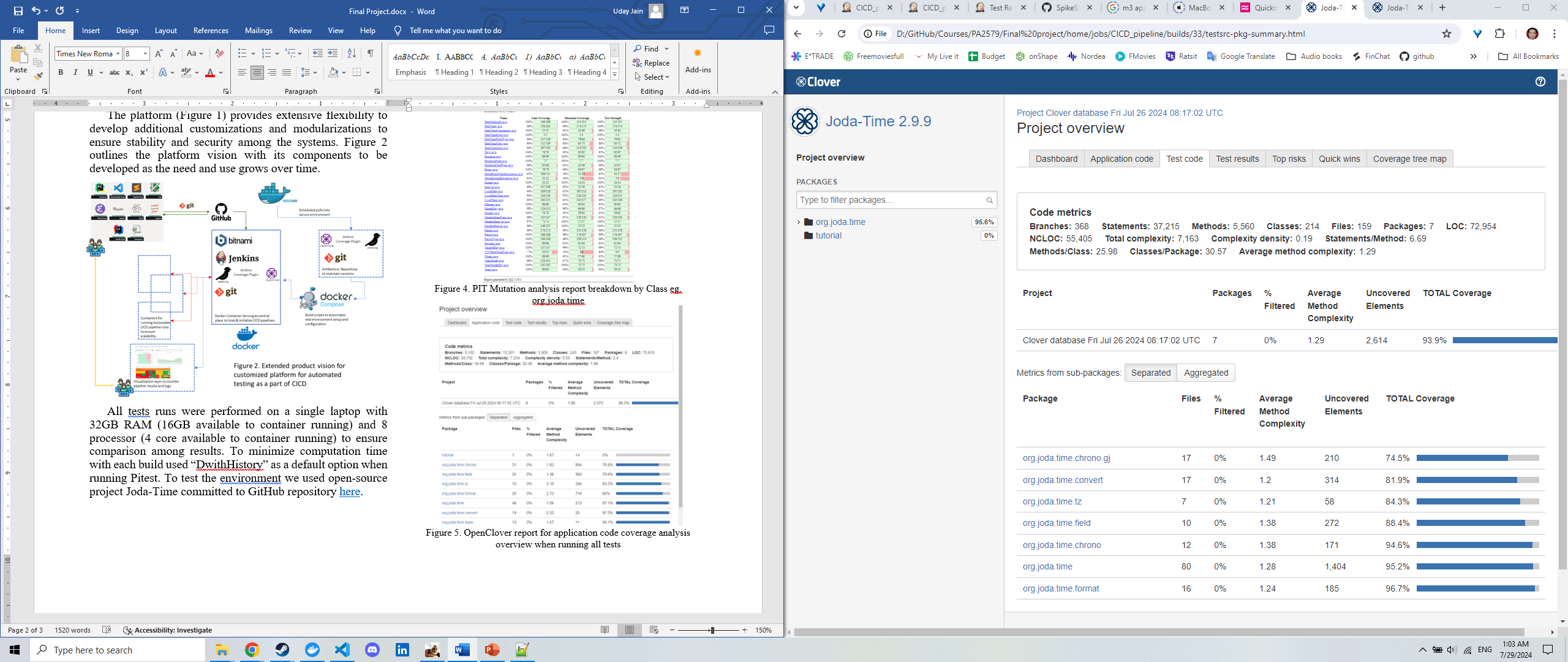


Figure 6. OpenClover report for test code coverage analysis overview when running all tests

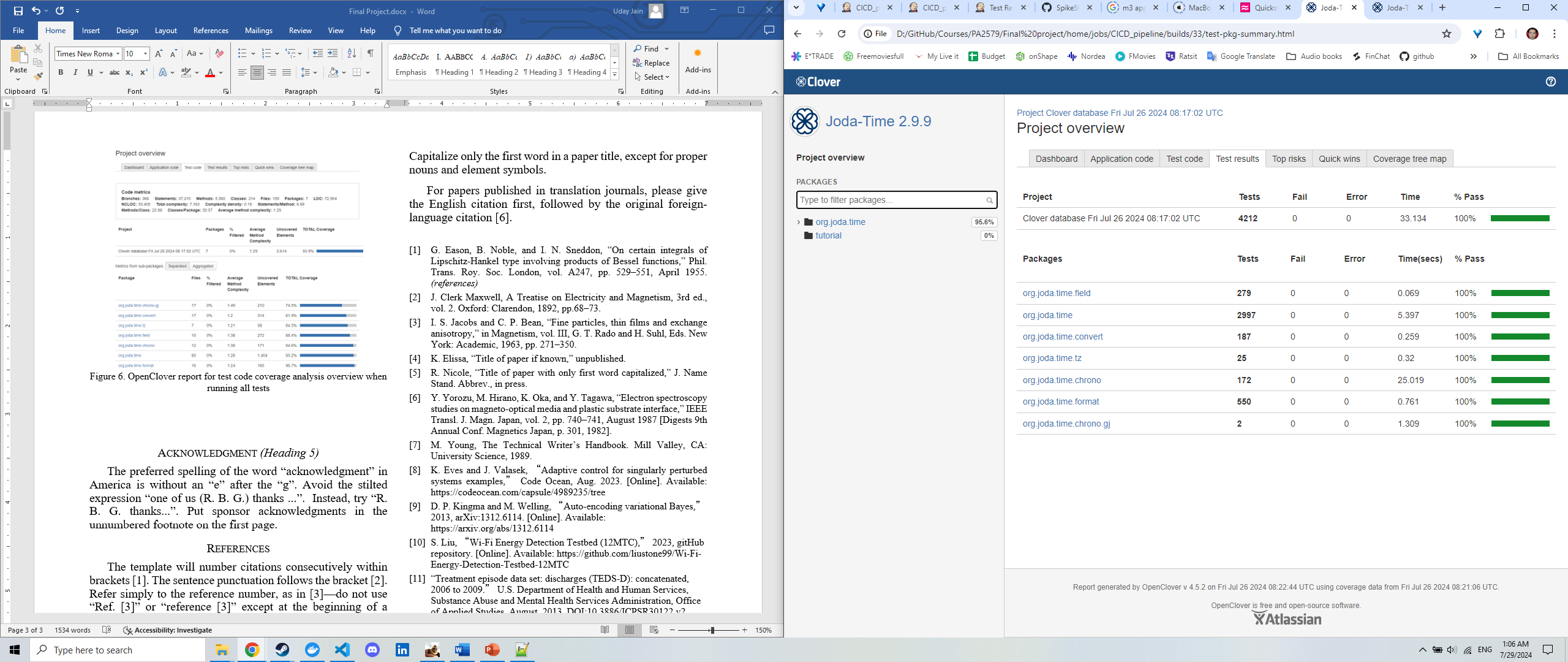


Figure 7. OpenClover report for test results when running all tests

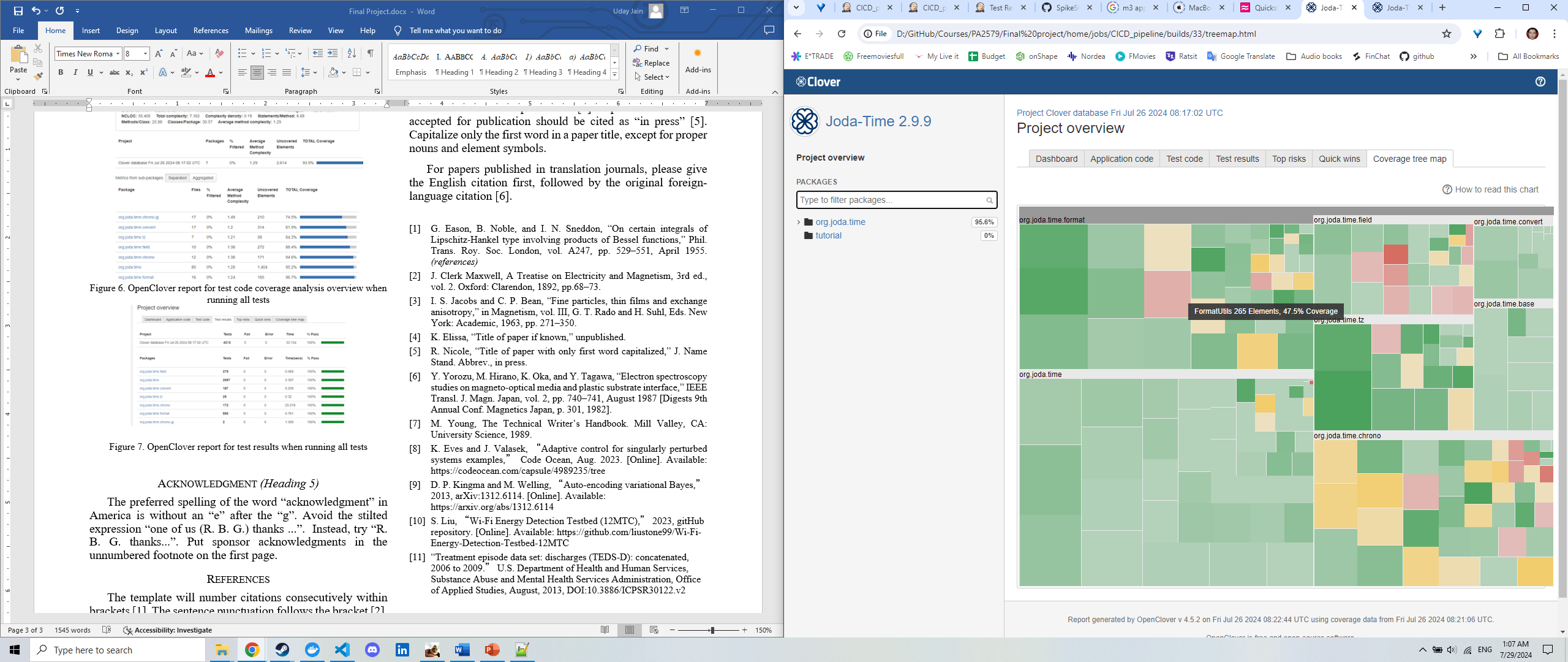


Figure 8. Coverage Tree map from OpenClover report when running all tests

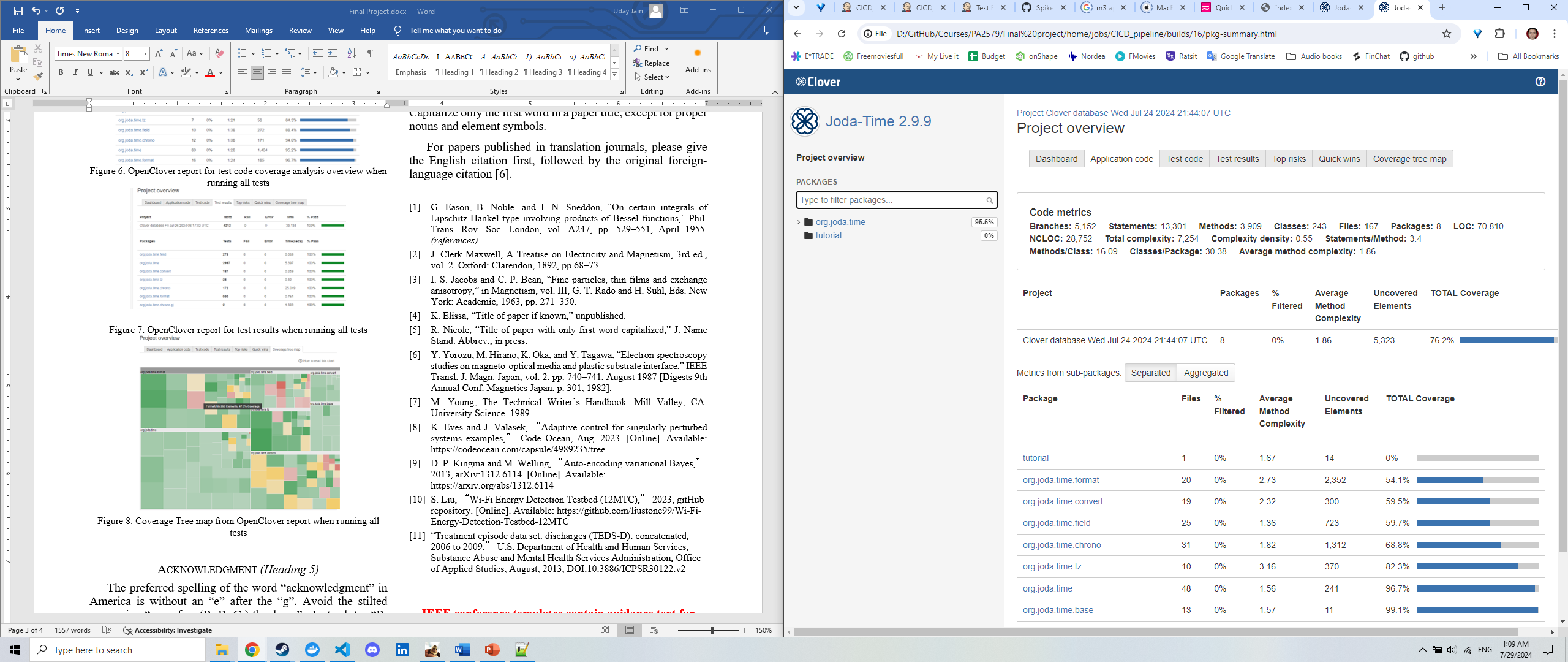


Figure 9. OpenClover report for application code coverage analysis overview when running only *org.joda.time.TestAll.suite()*

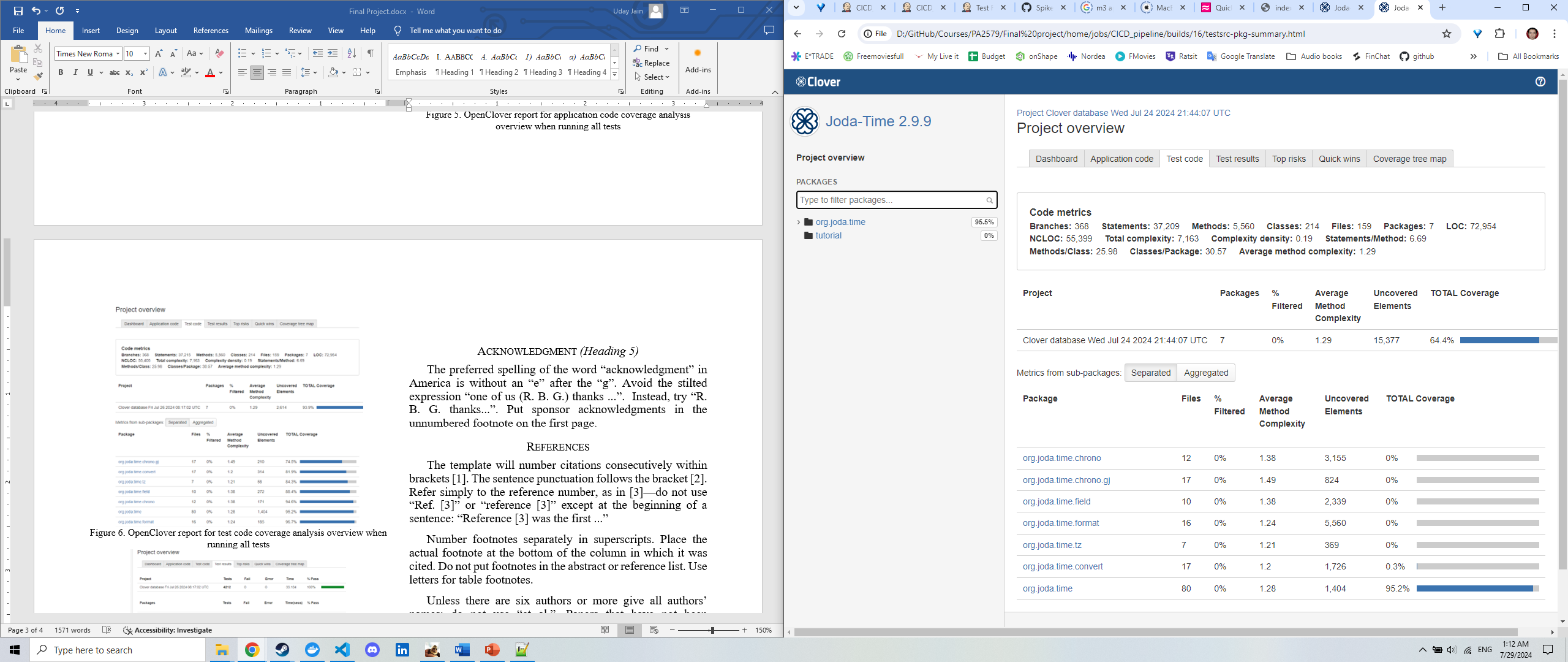


Figure 10. OpenClover report for test code coverage analysis overview when running only *org.joda.time.TestAll.suite()*

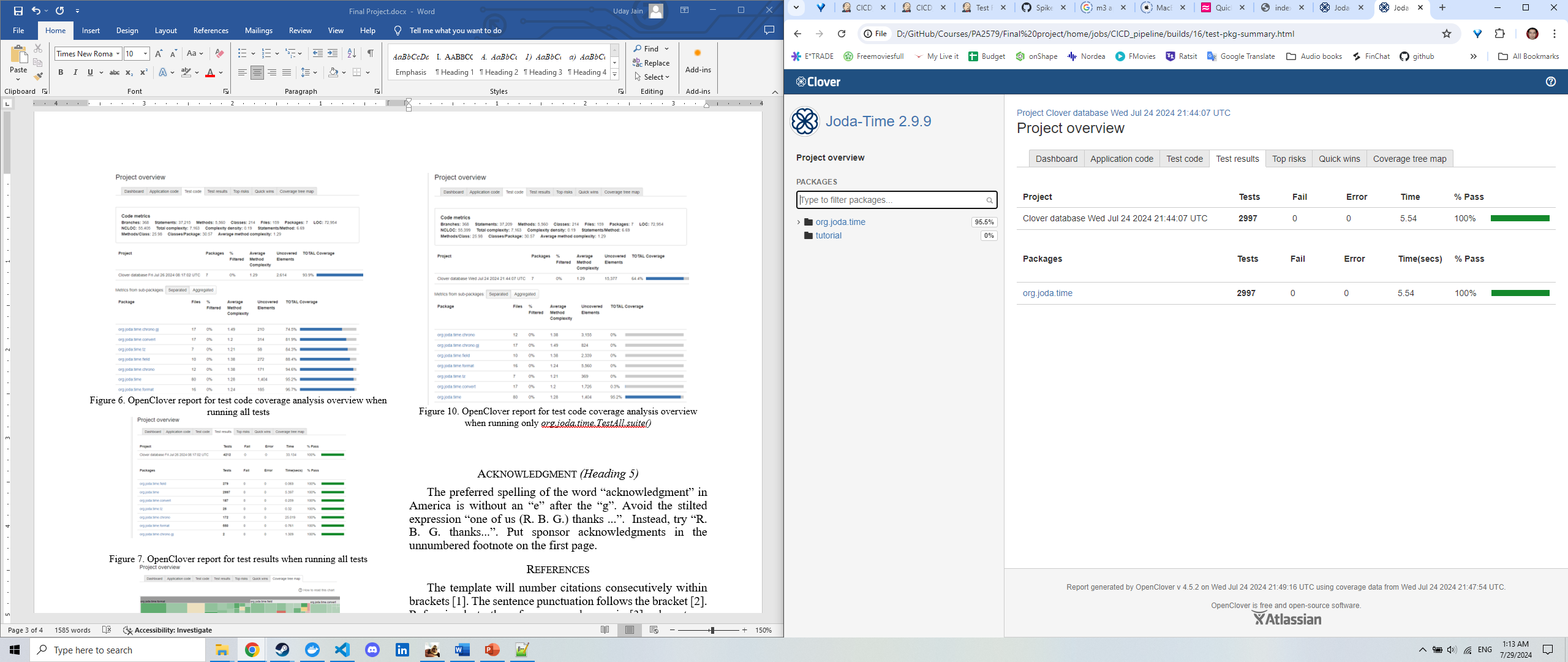


Figure 11. OpenClover report for test results when running only *org.joda.time.TestAll.suite()*

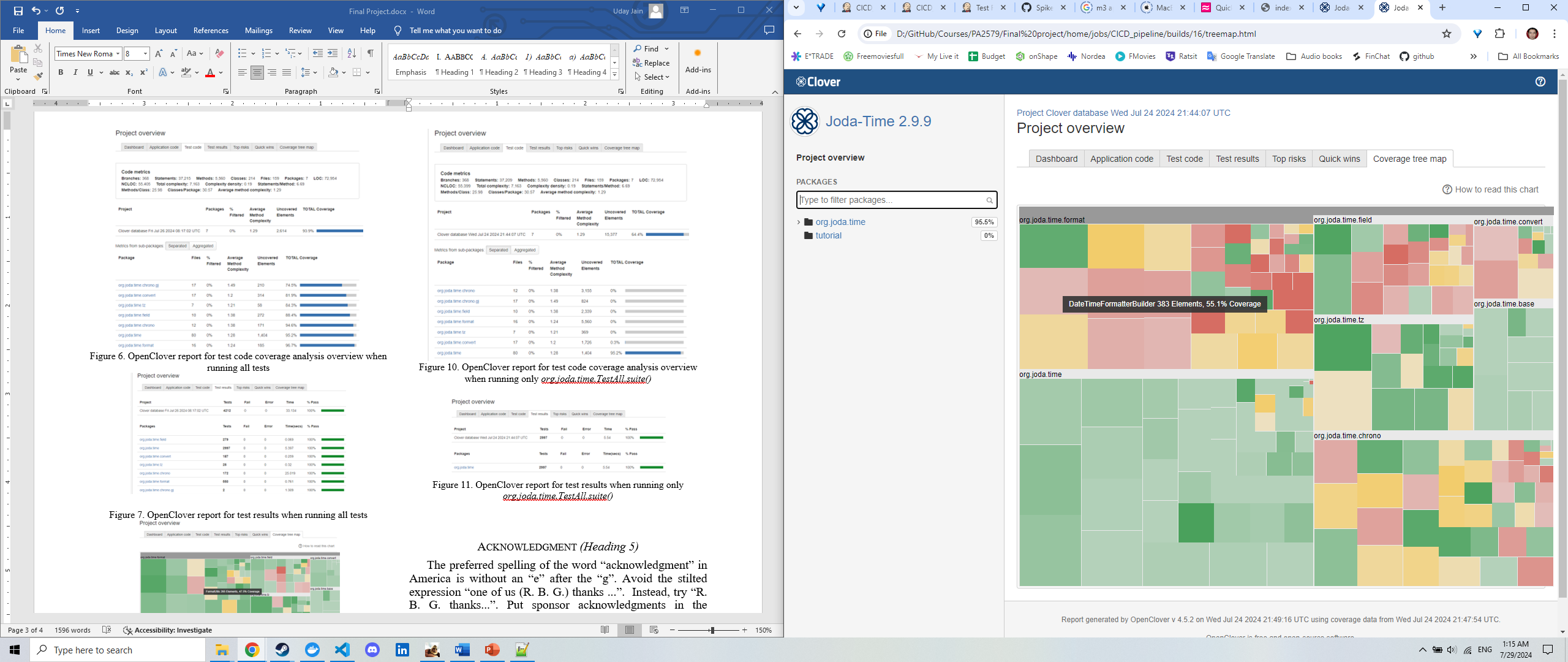


Figure 12. Coverage Tree map from OpenClover report when running only *org.joda.time.TestAll.suite()*

# Discussion

The automated test environment provides a centralized solution to automate analysis of testing done as a part of product development. The environment encourages the developers to use key metrics with standardized definitions such as % Code Coverage, % Test passed, % Mutation Coverage and % Test Strength to evaluate their commits. This helps development team adopt principals of LEAN software into their Agile sprint cycles.

The generated reports provide quantitative (using summary metrics) and continuous feedback that is valuable to the development team as well as PMs when developing patches to existing software products. As observed from figures 10 and 6, the total test code coverage has improved from 64.4% to 93.9% once we included all test suites into our development code. Metrics like these provide crucial insights into quality of our code, providing continuous feedback. Furthermore, availability of such centralized and standardized metrics helps review code and enable face to face discussions within respective development teams to improve solutions providing customer value in terms of stability and minimized defect probability.

Careful choice of metrics and reporting formats can enable simple and scalable solutions to automated testing that can be adopted across the wider organization and development teams.

Detailed reporting views such as those represented in Figure 5-12 can provide insights into classes and packages that require attention enabling the team to practice continuous improvement overtime through summarized tracking across multiple builds. Using observations from tree map in figure 12, one can easily identify package (*org.joda.time.format*) to be prioritized for test cases to improve % test coverage. One can further identify class (DateTimeFormatterBuilder) that can have the biggest impact on the summary metric (% test coverage). The OpenClover report further provides details into subclasses as well as location within overall code to help development team get started, enabling members to be self-organized and independent.

PIT Mutation reports provide similar level of granularity and flexibility to deep dive, enabling teams to rally across summary metrics such as % mutation coverage and % test strength. This helps development teams to ensure not only test coverage but also strength of test suits to capture borderline scenarios. Like OpenClover reports the Pit Mutation Reports help users to deep dive into class level analysis to identify classes that require additional tests to ensure high test strength.

A combination of % test coverage and % test strength can also help development teams track redundancy in test suits and minimize irrelevant test cases. This could potentially serve as a part of test suit optimization. This methodology coupled with other test suit optimization techniques such as test case prioritization, test suit minimization and test suit selection can play a crucial role into reducing resource requirements including both computational resources and time.

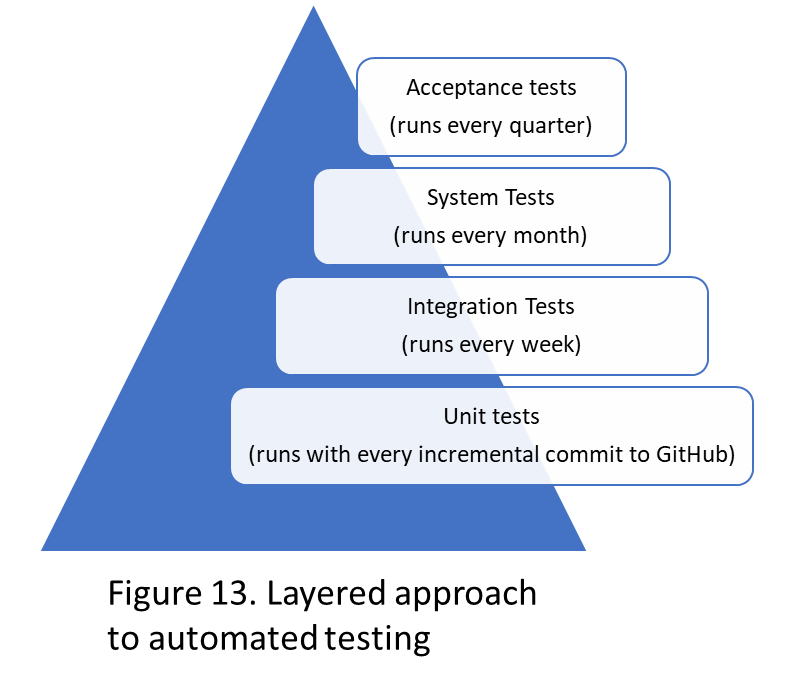
While the automated testing environment provides a scalable centralized solution that can be easily adopted by any development team and scale resources based on needs (owing to containerized solution), it may require additional levels of customization as practices and product requirements evolve over time. Different teams may find other metrics more relevant given their stage in product development life stage. We also found both OpenClover and PIT Mutation Report to be very exhaustive and detail intensive, making it difficult to digest in a fast-moving product development cycle.

As a part of the minimum viable product, we excluded any limitations on summary metrics to mark build as a failure. This is representative of the difficulty one would face when orchestrating such conditions across a diverse set of products and teams. In practice, it may be rare to find all or many products at same/similar life stage and thereby making it difficult to impose thresholds on key metrics at scale and such thresholds needs to be customized for each project. To facilitate this process, as shown in Figure 2, we recommend isolating the visualization layer to provide higher level of flexibility to reporting as well as integrating reports directly into end-customer or organization wide solutions.

During the exercise we faced many challenges identifying right combinations of plugin and package versions. This may remain a never-ending problem especially when using a diverse set of open-source solutions that are often released and developed at varied paces. To ease maintenance efforts, we recommend planning development of an antifactory store that stores relevant versions of the key components. This feature also helps maintain a closed system with high levels of security.

While the MVP only uses a single docker container, different projects may need different levels of computation and storage resources. To efficiently manage the available resources a more distributed architecture is recommended. As shown in figure 2, we recommend splitting the main Jenkins control panel and project builds into separate containers, with build containers that spins up based on requirements.

While PIT mutation analysis provides a thorough analysis of the tests, it is very time consuming and has a major impact on both the computational resource requirements as well as the run time. The Pitest package allows customization of the analysis to our needs using many parameters such as withHistory, targetClasses, targetTests, excludedMethods, excludedClasses, excludedTests, excludedTestClasses and maxMutationsPerClass. The current setup ran out of resources and crashed when default and unrestricted configurations were used. Given the resourcing limitations of the system we used maxMutationsPerClass during our test runs to limit need for compute resources. Setting these values for all projects may be difficult and would still require participation from developers. Care must also be taken to ensure abuse of such levels of customizations since it may lead to biased top level summary metrics. There may be a need to establish standard recommendations to allow such customizations for individual projects and consequently adequate training to within each product unit to familiarize developer of ways of working. Similar customizations are also available within OpenClover and other packages.



Although not implemented and tested as a part of the process, based on the feedback from the development community, we recommend a layer approach to structing testing across the projects builds. Figure 13 represents a pyramid structure with multiple smaller unit tests that run with every incremental commit to the GitHub repository at the bottom and larger acceptance tests or system tests that run every month or quarter. Such levels of customizations can easily be achieved via splitting the project into multiple projects with each scheduled to run at required frequency. Each project can now be configured to produce relevant level of reporting, allowing us to track low level metrics such as % code coverage for unit tests while high level metrics such as average response time for system tests. A separate visualization layer, as shown in figure 2, would also allow us to extract relevant pieces across these projects to provide reporting suited at different levels organization and customer needs. This approach can also help us minimize repetitive test runs with every build and save on precious resources.

Segregating reports can help development teams structure regular discussions within different forums such as a squad level detailed discussion every sprint or a product level discussion among PMs and EMs (engineering managers) from different subproducts every quarter to share insights from integration and system wide tests.

##### Conclusion

Even with all issues one may face when scaling, owing to effort required to setup and maintain the environment for automated testing, it may provide high returns when released centrally and adopted across multiple development teams. Organisational structures can easily be integrated into the layered structure of project to ensure standardized reporting solutions well suited across the diverse forums. This enables teams to adopt Lean principals and influence sprint management based on quantitative metrics, reducing effort towards analysis of results and adding incremental customer value with each sprint cycle.

As explained previously, the use of docker containers can offer efficient management of resources through both vertical and horizontal scaling capabilities. This flexibility however, do not offset the gains one can achieve through focussed and continuous approach towards tests suit optimisations. The former can only provide an initial breakthrough only to find system outgrow its resourcing capacity as inefficient project build and repetitive tests grow exponentially overtime. We must try to ensure tech stack efficiencies are coupled with efficiencies that can gained through adopting best practices and test suit optimisation within the development team.

Even with standardised solutions, a need for high level of flexibility and customisation when used across teams remains to capture edge use cases that may vary across individual units. Consequently effectiveness of such solutions equally depend on efficiency of an organisation and its people to adopt standardized practices for customizations and minimise abuse leading to biased metrics, unintended behaviours and regular discussion at multiple levels to utilise its complete potential in improving test analysis across the wider set of development teams.

##### Acknowledgment

Pending.

##### References

Pending.

1. G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*
2. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

1 Nidagundi, P., and Novickis, L.: ‘Introduction to Lean Canvas Transformation Models and Metrics in Software Testing’, Applied Computer Systems, 2016, 19, (1), pp. 30-36

2 Alwardt, A.L., Mikeska, N., Pandorf, R.J., and Tarpley, P.R.: ‘A lean approach to designing for software testability’. Proc. 2009 Ieee Autotestcon2009 pp. Pages

3 Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., and Juristo, N.: ‘A dissection of the test-driven development process: Does it really matter to test-first or to test-last?’, IEEE Transactions on Software Engineering, 2016, 43, (7), pp. 597-614

4 Fucci, D., and Turhan, B.: ‘A replicated experiment on the effectiveness of test-first development’, in Editor (Ed.)^(Eds.): ‘Book A replicated experiment on the effectiveness of test-first development’ (IEEE, 2013, edn.), pp. 103-112

5 Böhme, M., and Paul, S.: ‘On the efficiency of automated testing’. Proc. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering2014 pp. Pages

6 Dudekula Mohammad, R., Katam Reddy Kiran, M., Petersen, K., and Mantyla, M.V.: ‘Benefits and limitations of automated software testing: Systematic literature review and practitioner survey’. Proc. 2012 7th International Workshop on Automation of Software Test (AST)2012 pp. Pages

7 Milojkovic, J., Ciric, V., and Rancic, D.: ‘Design and Implementation of Cluster Based Parallel System for Software Testing’. Proc. 2020 Zooming Innovation in Consumer Technologies Conference (ZINC)2020 pp. Pages

8 Chuchuen, Y., and Rattanaopas, K.: ‘Implementation of Container Based Parallel System for Automation Software Testing’. Proc. 2021 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunication Engineering2021 pp. Pages

9 Poth, A., Werner, M., and Lei, X.: ‘How to Deliver Faster with CI/CD Integrated Testing Services?’: ‘Systems, Software and Services Process Improvement’ (2018), pp. 401-409

10 Wolf, J., and Yoon, S.: ‘Automated testing for continuous delivery pipelines’, in Editor (Ed.)^(Eds.): ‘Book Automated testing for continuous delivery pipelines’ (2016, edn.), pp.

11 Bitnami: ‘Popular applications, provided by Bitnami, containerized and ready to launch.’, in Editor (Ed.)^(Eds.): ‘Book Popular applications, provided by Bitnami, containerized and ready to launch.’ (GitHub, 2024, edn.), pp.

12 Polo, M., Tendero, S., and Piattini, M.: ‘Integrating techniques and tools for testing automation’, Software Testing, Verification and Reliability, 2007, 17, (1), pp. 3-39

13 Jenkins: ‘Discover the 1900+ community contributed Jenkins plugins to support building, deploying and automating any project.’, in Editor (Ed.)^(Eds.): ‘Book Discover the 1900+ community contributed Jenkins plugins to support building, deploying and automating any project.’ (2024, edn.), pp.

14 Ma, Y.S., Offutt, J., and Kwon, Y.R.: ‘MuJava: an automated class mutation system’, Software Testing, Verification and Reliability, 2005, 15, (2), pp. 97-133

15 Puri-Jobi, S.: ‘Test automation for NFC ICs using Jenkins and NUnit’. Proc. 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)2015 pp. Pages

16 Khari, M., Kumar, P., Burgos, D., and Crespo, R.G.: ‘Optimized test suites for automated testing using different optimization techniques’, Soft Computing, 2017, 22, (24), pp. 8341-8352

17 Sneha, K., and Malle, G.M.: ‘Research on software testing techniques and software automation testing tools’. Proc. 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)2017 pp. Pages

18 Stouky, A., Jaoujane, B., Daoudi, R., and Chaoui, H.: ‘Improving Software Automation Testing Using Jenkins, and Machine Learning Under Big Data’: ‘Big Data Technologies and Applications’ (2018), pp. 87-96

19 Cai, X., and Lyu, M.R.: ‘The effect of code coverage on fault detection under different testing profiles’. Proc. Proceedings of the first international workshop on Advances in model-based testing - A-MOST '052005 pp. Pages

20 Fleck, P.: ‘Utilizing Code Coverage Density to Enhance Software Quality Management Decisions’, Wien, 2022

21 Sánchez, A.B., Parejo, J.A., Segura, S., Durán, A., and Papadakis, M.: ‘Mutation Testing in Practice: Insights From Open-Source Software Developers’, IEEE Transactions on Software Engineering, 2024, 50, (5), pp. 1130-1143

22 Parsai, A., and Demeyer, S.: ‘Comparing mutation coverage against branch coverage in an industrial setting’, International Journal on Software Tools for Technology Transfer, 2020, 22, (4), pp. 365-388

23 <https://pitest.org/>

24 <https://openclover.org/>