QEX: Automated Testing Observability and QA Developer Experience Framework

Huang Luohua Locke*, Yap Kai Ting Keshia, Joseph Chu, Hock Yao Chua
Shopee Pte Ltd
5 Science Park Dr, Singapore 118265
{luohua.huang, keshia.yapkt, joseph.chuky, hockyao.chua}@shopee.com

***Abstract*—Automated testing is now applied on a large scale, especially in organizations that embrace a continuous integration and continuous delivery (CI/CD) software development process [1], [2]. While numerous studies have demonstrated the impact of automated testing on software development, there is currently no standardised industrial framework for providing a comprehensive high-level overview on testing activity across projects. This is vital for developing and maintaining test frameworks, and thereby a crucial part of the Quality Assurance (QA) *Developer Experience* [3]. In this paper, we present QEX, which is a model for integrating common testing data sources to obtain observability on test development in a way that provides transparent and actionable insights to relevant stakeholders. It is most suitable for enterprise contexts where testing is performed on a large scale. We provide an open-source implementation that is used at our company, together with an evaluation of its performance and possible ways to extend it further.**

***Index Terms*—Observability Framework, Testware Maintainability, Software Testing, Automated Testing, Software Metrics, Quality Metrics, Quality Evolution, Developer Experience, Monitoring**

I. Background

The frequent release of functional product code is heavily reliant on automated regression testing. In large-scale internet software companies, it is common for the number of test cases executed daily to rack up quickly [1], [2]. With such high usage, it is paramount to support QA engineers in their creation of complex test infrastructure. Test frameworks have to be developed and maintained in tandem with the frequent changes in product code to ensure the smooth functioning of the continuous integration and continuous delivery pipeline, and may sometimes be as complex as the product code that is shipped out.

Numerous advancements have been made in both research and industry sectors to support the software engineer's *Developer Experience* [3], [4]. While QAs may benefit from such tools, the daily work and expectations for QAs differs from developers enough to make it worthwhile discussing QA-specific productivity metrics. There is currently insufficient research conducted and tools created catered specifically for QA, which contributes to the perception of software testing as second-rate to software development [5], even though the maturity of the testing framework directly impacts production speed and efficiency.

*Corresponding author

Similar to production metrics that support the software engineer's Developer Experience, QAs can benefit from an observability system on their test automation framework. This can help chart the evolution of their test framework and improve it towards test automation maturity [6]. The state-of-the-art test automation frameworks that provide observability are too rigid as they are limited to offering analyses on tests written within their framework and make little mention of the impact of regression testing on the broader software development life-cycle. A good observability framework should be able to provide reliable metrics not solely on test automation development and execution, but also on the impact of test automation on the speed and quality of software development. It should also be flexible enough to accommodate QAs in their selection of the best test tools for each product over time, in view of the fast-changing nature of technology.

We present QEX, a monitoring framework that integrates common testing data sources for obtaining high observability on test development that is suitable for large-scale in enterprise contexts. It supports the QA engineer's Developer Experience [7] by providing comprehensive, reliable and timely feedback on their testing activity throughout the test automation development life cycle.

In this paper, we explain the design of the QEX framework and describe our open-source implementation of QEX that is currently used at our company [8], [9], providing the reader with handles to implement and customise it to suit their own needs. Finally, we provide an evaluation of our framework and discuss possible advancements to our implementation.

II. Preliminary

We will use the following definitions in this paper:

- *QA:* Quality Assurance, Quality Assurance Engineer, or Software Quality Assurance Engineer. For simplicity, we refer to any engineer who performs QA work as a QA engineer;
- *AT:* Automated Testing, automated test(s), or automated test case(s);
- *Developer:* Refers to a person who writes code, be it software or testware;
- *Sign-off:* Refers to a QA acknowledging that they have reviewed and tested the application, and are marking it as ready for release;
- *API:* Application Programming Interface;

- *Ticket:* Refers to report on a task management system, with details pertaining to its status, the particular problem or task being addressed, the reporter and other relevant details;
- *Test job:* Refers to an automated job that conducts the execution of a suite of automated test cases in a scheduled manner.

III. Approach

We first identified a list of metrics that help to quantify the usage, stability and development of automation test frameworks. These are reflected in Table I.

Categories	Example Metrics
Quality	Pass rate of AT cases Time taken to run AT cases Flakiness of AT cases Coverage of service by AT cases Number of maintainers per AT case
Dependency	Test cases/Services with highly correlated fail rate
Usage	Rate of execution of test cases Main executor of test cases
Development	Number of new AT cases run daily Services with sparse AT support Frequency of AT code change
Team	Allocation of AT work
Value	Number of tickets signed off by AT Number of bugs found by AT
Contribution	Percentage of AT cases written by individual/team Number of new test cases created daily

TABLE I
Common AT Metrics

Such information is vital for the performance of the QA team as it provides them with actionable insights to improve their testing infrastructure, allows for better management of task and resource allocation and provides transparent metrics to chart AT progress towards AT maturity [6]. They can be broadly distinguished into two broad categories - those obtained directly from testing activity, and those obtained indirectly from testing activity:

1) **Test Development & Execution Statistics**
   This category includes data derived directly from test development and execution, such as the number of test cases developed, the number of test cases executed, test execution results, time of test execution, test case maintainer information and time of last update. This data can be further processed to yield more complex statistics such as pass rate trends, speed of test development, and test flakiness.
   Since these statistics are generated directly from the automation pipeline, they provide concrete insights into the quality and reliability of the testware. Monitoring such statistics over time and across services can help QAs identify larger issues like faulty product dependencies or test flakiness [10]–[12]. This category contributes mainly to the "cognition" and "conation" aspects of the Developer Experience as mentioned in Fig. 1 of [3] as it supports QAs with accurate measurements of their testing infrastructure.
2) **Downstream Statistics**
   This category includes data created in response to the execution of AT test cases. Examples include bug tickets created due to product code defects found using AT, and feature tickets that were signed off using AT for regression testing.
   This data is mainly used to help engineers evaluate the returns on effort invested to develop and maintain the AT framework. It helps to quantify the value of their AT contribution towards software development, and can lead to an increased sense of ownership and satisfaction in their work. This category of feedback pertains primarily to the "affect" and "conation" aspects of the Developer Experience (Fig. 1 of [3]).

While this list is not exhaustive, it serves as a good starting point to integrate other quality and productivity metrics. We will unpack more of this in Section VI.

IV. Framework

In this section, we explain how some of the metrics listed in Table I can be obtained and visualised. A typical testing methodology comprises of the following main components:

- ***Git repositories*** containing the test code, with information related to test names, merge requests and commit history;
- ***A task management platform*** for bug tracking and agile project management. Typically, task tickets requiring testing contain details of the product under test and other information related to feature sign off. Bug tickets should contain details related to bug resolution, related to tickets signed off and for the completion of feedback loop for tasks that can be completed with AT.
- ***A testing platform*** sets up the environment for executing the test code and then runs the tests. Typically, tests are periodically executed on a job scheduler and/or triggered by continuous-integration (CI) pipelines during code commits or the raise of merge requests.

We developed the QEX framework based on the above so that it can be used as widely as possible. In order to achieve ease of use, high scalability and high observability, our framework adopts the following:

1) ***Having an event-driven architecture:*** QEX is composed of several test data sources that send data in an event-driven manner to a centralised web service, which is stored in a time-series database and visualised on a graphical dashboard. An event-driven architecture pattern is used so that it can handle the stream of continuously-generated test data with minimal data loss, and be able to scale to include other data sources easily [13].
2) ***Obtaining data from the source:*** In order to get the most accurate information on testing activity, we need to obtain it as soon as possible after test execution. This can be done by modifying the testing library so that it sends test information to the database immediately after

each test run. In the case of other components that are less time-sensitive, a simple periodic querier can be used to fetch the data.

3) ***Integration of data:*** We collate all test-related information in a centralised location via a dedicated service. In this way, we can connect different sources of data to provide more meaningful notifications. For example, given that a git merge request includes task ticket information, and that it triggers a test job pipeline directly, we can use our dedicated service to directly update the related tickets with its regression test results. This allows the relevant stakeholders like project managers, release managers and developers to ascertain regression status directly from the task management platform, thereby improving the efficiency of the CI/CD process.
4) ***Ensuring versatility:*** Every component is constructed using reusable inter-operable modules communicating via open interfaces. This means that information can be collated from different sources easily, including legacy test infrastructure [14], [15]. This allows us to consolidate and display all relevant information in a single dashboard and thus provide a comprehensive picture of testing activity across multiple projects. This means that users are able to freely choose the best test tool for each product and enjoy unified observability.

V. Implementation

Our QEX implementation with instructions and sample configurations can be found at [8]. A demo video is available at [9]. Refer to Fig. 1 for a graphical representation of our QEX architecture. Users may use our implementation as a reference and use appropriate substitutions if desired.

A. Selection of tools and interfaces

The current implementation of QEX at our company makes use of the following state-of-the-art tools:

- ***RESTful API:*** An open architecture representational state transfer based on HTTP protocol. A RESTful API server is used to assist in the processing and collection of test data in a flexible manner;
- ***InfluxDB:*** A high-speed time-series database engine for building analytic applications [16];
- ***Kafka:*** Refers to Apache Kafka, an open-source distributed event streaming platform [17]. We use it to improve reliability and minimise data-loss;
- ***Grafana:*** An open source analytics and monitoring solution [18]. It is capable of displaying the time-series data in meaningful ways with straightforward configurations;
- ***Jira:*** An issue/ticket tracking product developed by Atlassian that allows bug tracking and agile project management [19];
- ***GitLab:*** A version control system [20], used to store and manage AT code;
- ***Jenkins:*** An open source automation server for building and deploying. We use it to organise and run test jobs;
- ***Golang:*** An open source programming language with built-in testing features. It can be easily modified to integrate with our observability framework.

B. Collection and processing of data

All data traffic is handled through the *QEX Web Server* [8] and directed to a centralised database built using InfluxDB. The following describes how data is processed from the various sources:

1) **Git Repository Data**
   This refers to the test case information for all test cases stored in the master branch of the test repository. The test case information includes the test case name, last maintainer and latest commit ID, which can be easily obtained from git (refer to the bash script provided in [8]). The insertion of Git data into the QEX database is decoupled using a separate downstream Kafka module to prevent congestion in the pipeline. The following components in [8] handle this:
   a) *QEX Gitlab Querier* pulls data from Gitlab repositories to get information on the latest suite of test cases, identify the contributors of test code, and other merge request activity;
   b) the *QEX Git Log Message Queue* Consumer reads and pushes Git log data gathered upstream;
   c) the *QEX Jenkins Build Consumer* consumes messages about Jenkins job information from the *QEX Web Server*, then parses Jira ticket IDs related to the merge request, branch or commit that triggered the job on Jenkins; and update the Jira tickets with Jenkins build/deploy status and test results.
2) **Test Execution Data**
   We modified the Golang testing library [21] to send test case execution data to the *QEX Web Server* via a RESTful API call after every test run. This data includes the test case's product information, test job run identifier, case name, git branch, maintainer, timestamp, duration, and result. The modified version can be found at [22].
3) **Test Job Data**
   The *QEX Jenkins plugin* [23] collects test results and other job information from Jenkins and sends it to the *QEX Web Server*. The information is parsed and uploaded to the database via the Kafka module mentioned above. The job information includes the job's trigger, related git information like code branches, commit messages or merge requests. The git information can be used to determine the set of Jira tickets associated with the test job, which are in turn updated by the *QEX Web Server* with the test results.
4) **Task Management Statistics**
   The *QEX Jira Querier* [8] gathers information about AT-related tickets from Jira obtains this information using Jira Query Language (JQL) queries and is scheduled to run regularly using a cron job.

C. Calculation of metrics

In this section, we outline a few statistics currently provided by QEX and possible inferences that can be drawn from their values. Some statistics, namely the *Active Score* and *Stability Score*, are newly-introduced in this paper as they have proven useful in our own industrial context. For the sake of precision, we use the mathematical set notation to explain the derivation of our test metrics. First, we present some definitions:

Definition 1. *Let K be the set of test cases in the Git repository. Each test case can be represented by the tuple (c, μ), where c is the name of the test case and μ is its Git commit ID (note: this can be associated with any Git branch).*

Definition 2. *Let $\Pi := \{0, \pm 1\}$ be the set of possible test results, where $1, 0, -1$ indicates a pass, skip and fail respectively.*

Definition 3. *Given a Unix timestamp t, let $\mathcal{C}(t)$ be the set of test executions completed at time t with result ℓ, given by the set of tuples (c, μ, t, ℓ) where $\ell \in \Pi$ and $(c, \mu) \in K$.*

Definition 4. *Given a period of time $T \subseteq \mathbb{Z}$, we can define*

$$\mathcal{C}(T) := \bigcup_{t \in T} \mathcal{C}(t)$$

as the set of test executions completed within the time T.

We now proceed to define some of our key derived metrics in QEX. Let $T \subseteq \mathbb{Z}$ be the period of time over which test cases in K are being executed.

1) ***Test Execution of** K **over** T* is the number of test cases in K executed in during the time T, given by

$$\mathcal{N}(K, T) := \#\{(c, \mu, t, \ell) \in \mathcal{C}(T) | (c, \mu) \in K\}.$$

It can indicate the amount of usage of AT.

2) ***Active Score of** K **over** T* is the ratio of the number of test cases in K that are executed during time T to the number of test cases in K that are in the latest version of the master branch, given by

$$\alpha(K, T) := \frac{\#\{c | (c, \mu, t, \ell) \in \mathcal{C}(T), (c, \mu) \in K\}}{\#K'}.$$

where $K' \subseteq K$ is the test cases in the latest version of the master branch of K (used for the nightly run). An active score of 0 indicates that no master branch test cases in K were executed during time T, while an active score of 1 would mostly likely indicate that every test case in the master branch was executed. An active score exceeding 1 mostly likely indicates test execution from non-master branches. This score highlights the importance of maintaining a slim test code repository with minimal dead code, and helps QAs assess the level of usage of AT in a product line

3) ***Pass rate of** K **over** T* is the percentage of test cases in K that passed during time T, given by

$$\zeta(K, T) = \frac{\#\{(c, \mu, t, \ell) \in \mathcal{C}(T) | \ell = 1, (c, \mu) \in K\}}{\#\mathcal{C}(T)}$$

if $\mathcal{C}(T)$ is non-empty, or 0 otherwise. A pass rate of 0 means that no test cases passed or that no test cases were executed. A pass rate of 1 indicates that every execution of every test case in K passed during T.
Note: skipped test cases are also considered tests that have been executed in our framework. This convention can easily be modified depending on user preference. *Skip rate* and *Fail rate* metrics can be defined similarly.

4) **Stability Score of K over T** is a compound metric, given by

$$\gamma(K, T) := \alpha(K, T) \times \zeta(K, T).$$

A stability score of 0 means that either all master branch test cases in K were skipped, failed, or not executed during time T. A stability score of 1 most likely indicates that test cases executed during the nightly runs have all passed. A stability score exceeding 1 most likely indicates test execution in non-master branches with a high pass rate. Test case stability is paramount for effective AT as it can flag issues and bugs in the product code with a high true positive.

5) **AT Sign-off of K over T** is the number of tickets signed off due to the test execution results of at least one test case c where $(c, \mu) \in K$ for some $\mu \in \mathbb{Z}$ in the period T. A higher number implies that the AT in K has more impact on the software development continuous integration and continuous development process.

6) **Bugs Found by K over T** is the number of tickets created due to the test execution results of at least one test case c where $(c, \mu) \in K$ for some $\mu \in \mathbb{Z}$ in time T. A spike in this score may flag critical software issues.

These metrics provide a basis for obtaining key insights into the test development life cycle. While we have provided possible interpretations for individual scores, trends, and test statistics, the user should draw their own conclusions in light of their testing context and the test metrics as a whole.

D. Visualisation & Monitoring

The *QEX Grafana Dashboard* [8] displays real-time test metrics in an intuitive and interactive manner. Fig. 2 presents an overview of the dashboard from which stakeholders are able to get a grasp of the testing system quality at different levels of granularity. The user can easily filter metrics by product line, QA and time period by directly toggling drop-down lists on the dashboard.

VI. Evaluation

A. Performance

In our organisation, QEX currently supports over 100 QA engineers across 10 product lines, with tens of thousands of tests being executed daily.

Each of the components are integrated to a common notifications system that will alert the relevant engineers whenever there is an error requiring manual intervention, thereby allowing for quick action to recovery. There is no significant impact on testing caused by the addition of the QEX monitoring

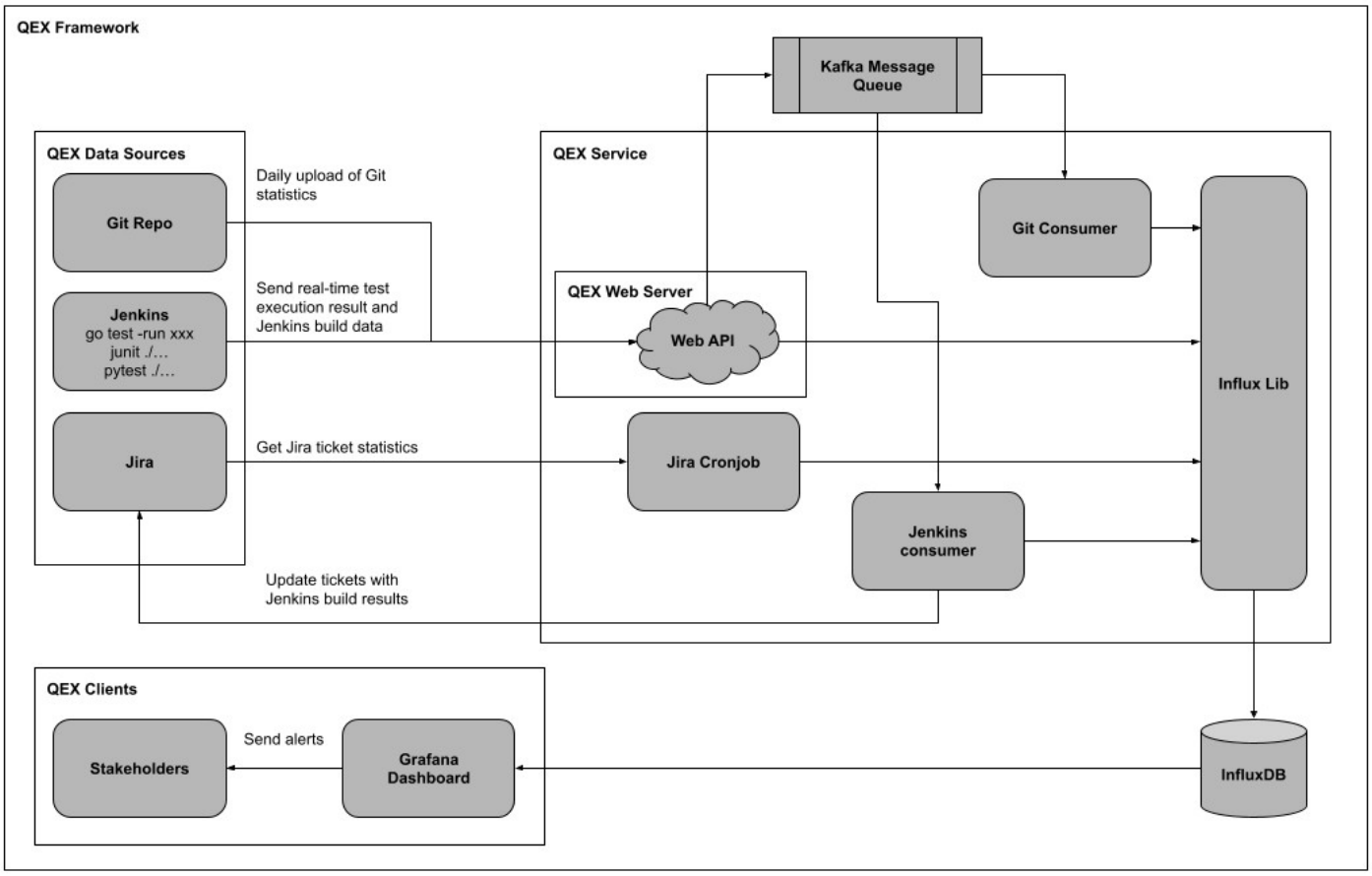

Fig. 1. Architecturural schema of QEX. Data is first pulled directly from the data sources, processed by the various components of the QEX Service, and finally stored in the QEX database to be presented to the QEX clients.

layer in the different components as the modifications are rather lightweight. Due to our event-driven architecture, our implementation is sensitive to changes in the test framework and test activity with an average reaction time of a few milliseconds. The integration of the Kafka module also helps to minimize data loss. The design of QEX allows its users to easily plug in additional data sources as well as scale individual services as required.

B. Challenges faced

One design challenge faced is in the selection of metrics. We acknowledge that every set of metrics will display a certain bias, and so strove to select a variety that can provide a good handle for testing observability that would be meaningful to QAs and other stakeholders with regard to test execution and impact on the software development life cycle. This provides a starting framework that can be easily supplemented to include other important metrics like code coverage by AT and progression of bug ticket statuses caught by AT, in accordance with the management direction.

Another limitation is that our implementation is not context-sensitive and requires some level of manual flagging during edge cases. For example, as the maintainer of a test case is defined by the git log command to be the last contributor to the test function, QEX may display an inaccurate representation of workload distribution after code refactoring. While the maintainer could be defined differently, any definition would have its own set of trade-offs. Users of QEX need to be mindful of these limitations when interpreting the metrics in context.

The framework provides simple metrics that can be easily obtained for a wide variety of projects. However, the trade-off for simplicity is that the metrics provided could be abused in ways that are counter-productive to improving the QA experience. It is not recommended to measure the quality of work produced by engineers from these metrics in an isolated manner without considering other quality metrics that are not as easily quantifiable, since not all tests and products are created equal. Teams that wish to implement QEX should be deliberate in communicating how QEX metrics relate to the

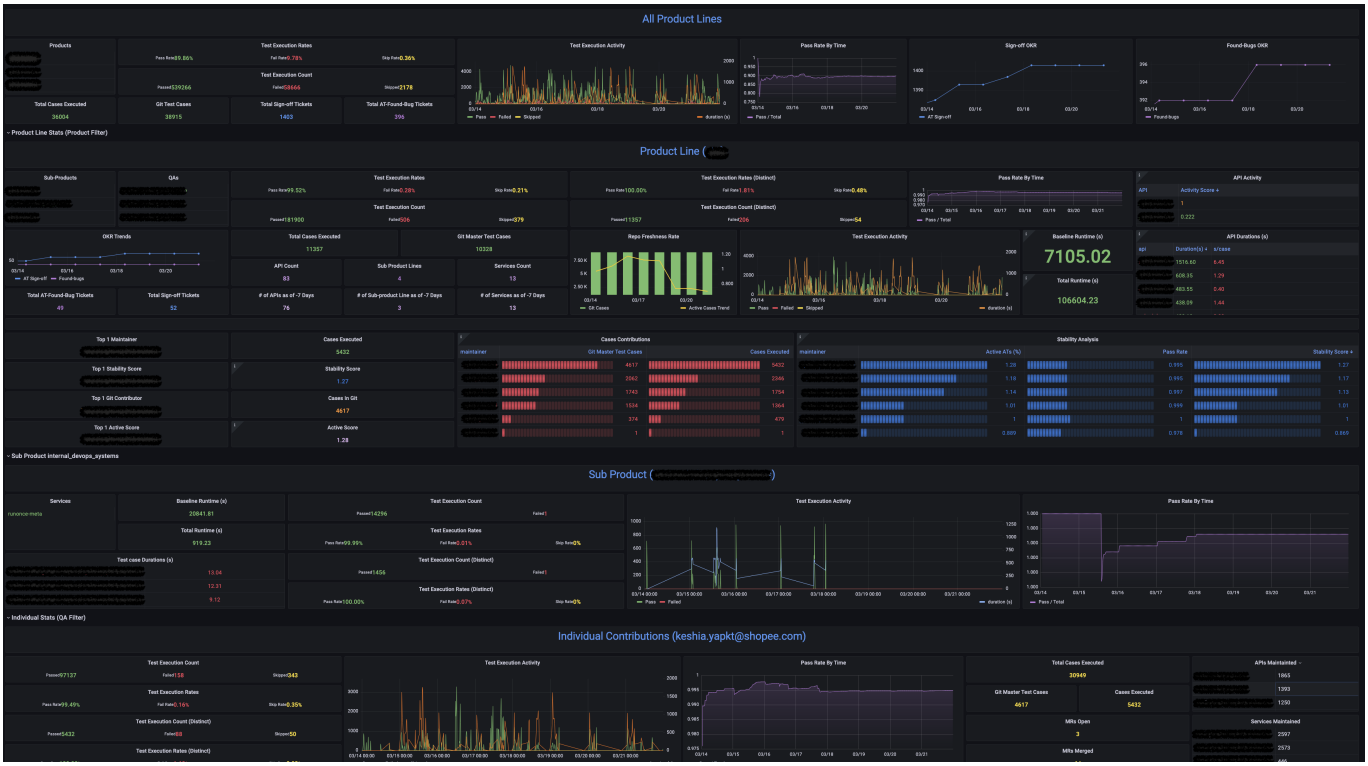

Fig. 2. A snapshot of the QEX dashboard which showcases AT data and analysis at varying levels, from an overview across product lines to individual contribution.

business goals of the team and how they will be utilised from a management perspective.

C. Barriers to adoption

1) Technical complexity: QEX comprises multiple components working together to provide a comprehensive and seamless experience for its users. Our approach was to streamline the deployment process while still allowing for flexibility in individual components. This requires a level of technical ability to deploy and maintain the different components. However, setting up QEX is a one-time effort and its payoff is significant when used in organisations where testing is conducted at scale.

2) Cost of using customised tools: We recognise that using a customised testing library requires extra maintenance effort during version updates. Moreover, while the modifications can be easily replicated in other testing frameworks like JUnit and Pytest, it may be more challenging to achieve the same results when using other frameworks, such as web-based testing. However, this customisation is necessary only to achieve the highest sensitivity and capture test data in real-time. Users may opt to send collated test data further down the pipeline if they do not mind the trade-off in sensitivity.

VII. Future Work

As QEX is built using an open architecture that can be easily scaled and integrated with other systems, there are many ways in which it can be enhanced through the integration of additional data sources. Distributed tracing libraries such as Jaeger and other code coverage tools can provide test case coverage and integration test metrics. The test data can also be further processed to provide alerts in the event of large scale failures, downtime of the testing infrastructure or other testing irregularities, and provide other quality metrics like the rate of bugs released into production and analyses of test flakiness. Confluence and SonarQube can provide data such as links to product code documentation and code quality metrics.

VIII. Conclusion

In this paper we have presented a framework that serves as a blueprint for designing integrated testing infrastructures with high observability that can elevate the QA Developer Experience. It is most suitable for enterprise contexts where testing is done at scale. We have provided an open source implementation of the framework with potential ways it can be extended and customised to suit different use cases by the community in the future. The metrics introduced in this framework can also act as a standard for measuring QA activity and involvement in future research studies.

References

[1] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242, 2017.

[2] John Micco. The state of continuous integration testing @Google, 2017.

[3] Fabian Fagerholm and Jürgen Münch. Developer experience: Concept and definition. In *2012 International Conference on Software and System Process (ICSSP)*, pages 73–77, 2012.

[4] Robert Chatley. Supporting the developer experience with production metrics. In *2019 IEEE/ACM Joint 4th International Workshop on Rapid Continuous Software Engineering and 1st International Workshop on Data-Driven Decisions, Experimentation and Evolution (RCoSE/D-DrEE)*, pages 8–11, 2019.

[5] Luiz Fernando Capretz, Pradeep Waychal, Jingdong Jia, Daniel Varona, and Yadira Lizama. Studies on the software testing profession. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 262–263, 2019.

[6] Yuqing Wang, Mika V. Mäntylä, Zihao Liu, Jouni Markkula, and Päivi Raulamo-jurvanen. Improving test automation maturity: A multivocal literature review. *Software Testing, Verification and Reliability*, n/a(n/a):e1804, 2022.

[7] Sigrid Eldh, Kenneth Andersson, Andreas Ermedahl, and Kristian Wiklund. Towards a Test Automation Improvement Model (TAIM). In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 337–342, 2014.

[8] Luohua Huang, Joseph Chu, Keshia Yap, and Hock Yao Chua. QEX: Automated testing observability and developer (QA) experience framework. https://github.com/luohuahuang/qex, 2022.

[9] Luohua Huang, Joseph Chu, Keshia Yap, and Hock Yao Chua. Qex demo. https://youtu.be/LkhGmEFfCgs.

[10] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444, 2018.

[11] Wing Lam, Krvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1471–1482, 2020.

[12] Martin Gruber and Gordon Fraser. A survey on how test flakiness affects developers and what support they need to address it. *15th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2022*, 03 2022.

[13] Tim Bray. AWS re:invent 2019: [repeat 1] moving to event-driven architectures (svs308-r1). https://www.youtube.com/watch?v=h46IquqjF3E, 2019.

[14] T. Sigwele, A. Naveed, Y.F. Hu, M. Ali, J. Hou, M. Susanto, and H. Fitriawan. Building a semantic RESTful API for achieving interoperability between a pharmacist and a doctor using JENA and FUSEKI. In *Journal of Physics: Conference Series*, volume 1376, 10 2018.

[15] Miroslav Bures, Bestoun S. Ahmed, Vaclav Rechtberger, Matej Klima, Michal Trnka, Miroslav Jaros, Xavier Bellekens, Dani Almog, and Pavel Herout. Patriot: IoT automated interoperability and integration testing framework. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 454–459, 2021.

[16] Influx Data Inc. Influxdata. https://www.influxdata.com/, 2020. V1.8.5.

[17] Apache Software Foundation. Kafka. https://kafka.apache.org/, 2021. V2.6.0.

[18] Grafana Labs. Grafana. https://grafana.com/, 2022. V8.4.3.

[19] Atlassian. Jira. https://www.atlassian.com/software/jira, 2022. V8.20.4.

[20] Gitlab Inc. Gitlab. https://about.gitlab.com/, 2022. V13.6.7-ee.

[21] Google, Open Source Comunity. Golang. https://go.dev/, 2022. V1.17.6.

[22] Luohua Huang, Joseph Chu, Keshia Yap, and Hock Yao Chua. Golang customised version for QEX framework. https://github.com/luohuahuang/go, 2022.

[23] Luohua Huang. Jenkins plugin for QEX. https://github.com/luohuahuang/qex-jenkins-plugin, 2022.