Supporting the Developer Experience with Production Metrics

Robert Chatley
Imperial College London
180 Queen's Gate
London, SW7 2AZ, United Kingdom
Email: rbc@imperial.ac.uk

***Abstract*—In continuous software engineering, systems are constantly updated and evolved, with small changes being deployed to production every day. These production systems can be a gold mine of data on how the system is operating. While application performance management (APM) tools are now commonly used in industrial practice to record such data, often in very large quantities, we see a need to convert this data into useful information, and to present it to developers within their regular development environment, in a context sensitive way. When a developer is working on a particular piece of code, we want to tell them how that particular piece of code is currently behaving in production. In this position paper we present a road-map of ideas for how new tools could be developed that harness this data, and present it back to developers in an actionable form, where they can use it to inform decisions about the impact of potential changes, as they work with the code day to day.**

I. Introduction

This paper presents a set of practical problems faced by developers working in a modern software engineering environment, delivering code changes to a production system iteratively and incrementally. The fact that we make continual small updates to a running system means that we have a large amount of data available to us about how that system is actually operating in production. We look at ways that this data could be harnessed to enhance the developer experience, and possible tools that we believe could be built to do this. We present a roadmap for a future research direction that we hope to follow over the coming months and years.

Predominantly we are thinking about web applications and systems of services, where typically many requests are served by the same components every day – in the case of high traffic consumer websites, perhaps millions of requests per day. This characteristic, together with Application Peformance Management (APM) software tracing and monitoring execution, gives us access to a wealth of data about how our software is behaving in production. The aim of the work proposed here is to harness this data in practical developer tooling, converting it into targeted, actionable, information.

II. Problem Areas

He we describe a set of practical situations in which we feel that presenting production data to developers in a suitable form *as they are working on the relevant code* may be helpful in improving their ability to understand, maintain, evolve, and fix problems with a running software system.

A. Exposing Execution Traces

As developers we have a mental model of how our code will execute in production. We may bolster our confidence with suites of unit tests to check the correctness of our code. These are helpful practices in building correct software, but what if our mental model of what happens in the software is different from what actually happens when it executes in the field. Consider the following function that we might have coded in a university application:

```java
public class StudentGrades {

  public void showRecords(User user) {
    if (user.role().equals("STUDENT")) {
      showSingleRecordFor(user);
    } else if (user.role().equals("STAFF")) {
      showAllRecords();
    } else {
      showGuestAccessPage();
    }
  }

  // more code here...
```

Here we might assume that the majority of people looking at their grades will be students, so most of the time we will enter the first block, and less of the time we will have staff looking up the grades for a whole class. We do not really expect guests to be accessing this system, as there is nothing for them to see here, but in case they do, we present them a friendly message pointing them to other relevant pages.

So, we might be surprised to find that in production, the most commonly executed branch is the third one. Why is that? A change in another part of the system has led to all the user roles being passed as lower-case strings, rather than uppercase. The system behaviour changed, but we would be unaware of it – probably later on we would get some help-desk requests from people saying they could not access their student records. There are no errors in the logs, so it may take us a while to track down the source of the problem.

What if we were able to overlay information about the execution paths that had been taken on top of the code, inside our IDE. Perhaps we could also highlight different branches in different colours to show which is the "hot" branch. If in this case, `showGuestAccessPage()` was highlighted in red, with a tool-tip popping up to show that 99% of calls to this method in the last 24 hours had taken this path, then we would quickly see the problem. A developer working on the codebase might even notice the problem before it is reported to the helpdesk. We envisage that building tools that overlay this kind of production data onto the code as the developers are working could increase the rate at which problems can be detected and fixed, as well as perhaps helping them to simplify code by removing features that are rarely used.

B. Reporting Exceptions

On discovering the above problem, perhaps we decide on a two-part change to improve the situation. Firstly, we make a case-insensitive match on the role attribute, so that the behaviour is corrected for staff and students. Secondly, we decide that really only staff and students should be accessing this system, it doesn't make sense to support guest users, so we decide to make this an exceptional state, and raise an appropriate error, with the hope that this will make problems more visible in future. The code is therefore updated as follows, committed and pushed to production. If the feedback cycle is short (and if we have sufficiently high traffic in production to yield significant results in a short time) developers may gain confidence in the success of their fix by observing the change in the metrics displayed after their revised code has gone live.

```java
public class StudentRecords {

  public void showRecords(User user) {
    if (user.role().equalsIgnoreCase("STUDENT")) {
      showSingleRecordFor(user);
    } else if (user.role().equalsIgnoreCase("STAFF")) {
      showAllRecords();
    } else {
      throw new UnauthorizedAccessException();
    }
  }

  // more code here...
```

A few weeks later, a member of the operations team is looking through some log files to search down a particular problem. They notice that quite a few of these `UnauthorizedAccessExceptions` have appeared in the log over the past few days. This is not related to the problem report they are working on, so they make a note of it in a Jira ticket to be discussed and possibly prioritised at the next sprint planning meeting, the following week. No-one else looks at these exceptions in the log, so the problem is not highlighted until that meeting, and at that point a task is created to "look into UnauthorizedAccess", but it is not marked as urgent, so it is put off to the following sprint.

During the current sprint, developers are working on the `StudentRecords` class, and so if they had known about this problem, they would be perfectly placed to investigate it quickly, but there is a disconnect in the flow of information – messages end up in a log file, which may be read by an engineer, which may lead to a ticket being created for developers to work on later. In fact what has happened is that a new class of student "POSTGRAD" has been introduced, who should also have access to their grades, but this is not being covered by the current code. It would be a quick fix if the developers were active in this piece of code anyway.

We propose additional tooling that would process production logs, and overlay statistics on to the code as the developer is working, showing how frequently exceptions have been thrown from each point in recent execution. In the example above, a red-highlighted line saying "Exception thrown 25,000 times today" might draw the developer's attention as they were working on additional functionality, and hopefully they would take the opportunity to investigate and fix the problem.

C. What Is Deployed Where?

In a continuous delivery process, it is common for teams to have several different pre-production environments through which their code passes before it goes live. There may be integration environments, where suites of automated acceptance tests run to check the technical correctness of the system, and then staging environments, where teams may assess whether or not the implemented features are correct with respect to business needs – are they what the customer wanted? After this, and possibly further stages, the code can pass to production.

With all of this complexity, and in a continuous software engineering environment where new changes are being made every day, we believe it would be useful for developers to be able to know when looking at a piece of code "is this code in production"? If a particular problem is reported in production at the beginning of the day, and a developer was investigating it this morning, is this piece of code that looks like a fix already in production? Or do we need to work more on this issue?

Working out what is deployed where is (or certainly should be) possible, given an automated build and deployment pipeline centred around structured version control usage, but it often is not obvious when looking at the code in the IDE. We propose tooling that would highlight the current environment in which each section of the code is deployed, whilst looking at it in the editor. If a developer opens a file, and there is a recent change that looks like it addresses (or perhaps causes) a particular problem, is that code currently in production?

A similar case might be where we commit and push a fix (perhaps the fix suggested in the previous section), and set it off in to the deployment pipeline to be tested and deployed. We have confidence in the reliability of the pipeline, so assume that after a short while the code will be deployed, and the users' experience improved. As we continue to work on the code, it would be useful for the highlighting in our editor to show if perhaps the code has reached the staging environment, but does not proceed to production. This might lead us to go and investigate what is blocking the pipeline – even if it is an unrelated issue. By adding appropriate telemetry we want to make it possible for developers to monitor the progression of each change through the pipeline, from within their regular working environment, without having to switch to another tool showing the dashboard of the deployment system.

III. Related Work

The idea of augmenting the display of code in the IDE with supplementary metrics data about the performance of the software has been explored in prior work by Beck [1], [2] and also by Cito et al in their work on *PerformanceHat* [3],

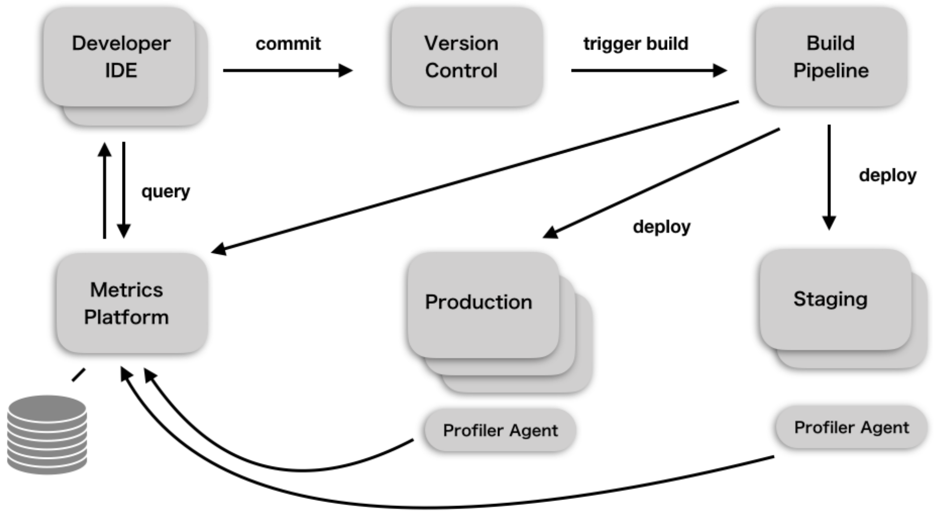

Fig. 1. Envisioned architecture for metrics gathering and analysis.

[4]. Both of these provide examples of using overlays and code highlighting inside the code editor window of the IDE to render context-specific performance data. In the case of Beck's work, much investigation has been done into how data can be presented visually in a useful way, making use of colouring and graphing of values, but the performance data itself is taken from profiling of the program during a performance test. In contrast, PerformanceHat builds on this approach to taking profiling information from execution of the software in a production environment, which is much closer to the situation that we envisage here.

We want to build on this work in a number of ways. Firstly, to monitor and present data not only to do with timings and performance, but also to expose the three cases described in Section II, which may require more fine-grained telemetry to be recorded and analysed. Secondly, we want to integrate more closely between the metrics and monitoring system and the build system. In a continuous software engineering environment, the version control and build systems hold a wealth of information about what change was made when, by whom, whether that version resulted in a successful build with all tests passing, and whether that build was then subsequently deployed into a particular environment. We want to close the loop so that this information can be used in order to aggregate and slice data from production execution – which log events correspond to which versions of the software?

IV. Challenges

We currently plan to build a system architecture similar to that shown in Figure 1. This figure shows the flow of a change from the developer's IDE, into version control, through the build pipeline, to staging and production. It also shows how the metrics platform is fed not only from profiling agents monitoring execution in both staging and production, but by the build pipeline itself. At one level there is an engineering challenge of effectively integrating these different tools and managing the flow of data between them. At a second level we want to investigate whether the types of metrics we can typically capture using APM tools can give deep enough insights in order to address the problems described above.

A. Tracking Changes

It is not normally true that once a commit is made, it is immediately running in production. Depending on the length of the build and release process, and whether there are any manual steps in this process, the time between commit and release to production could range from minutes to days (or even weeks). We therefore need integration with the build pipeline so that we can capture data about which version is deployed in which environment at which times, to give us accurate cross-correlation of information.

One option is for the build pipeline to push data to the metrics service every time that a new build is deployed, recording the build number, and perhaps the git commit hash that this corresponds to, so that monitoring data can be tied back to a particular deployed version later on. An alternative approach would be to make the monitoring agents and logging code aware of the currently deployed version, so that log events could be tagged at source with a particular version number. Which of these would be more effective is an open question.

Another issue around tracking changes is that as the rate of release increases, with the smaller batch sizes that continuous delivery favours, do we risk reducing the amount of data that we have about any individual version. If we did 100 deployments per day, each would only be live for a matter of minutes. In this state perhaps it will be more meaningful to aggregate the monitoring data over the time since a particular piece of code was last changed. This approach would make the analysis rather more intricate.

Another question is whether in presenting data perhaps rather than reporting current values (this exception was thrown 100 times today), it would be better to report changes and deltas (this exception was thrown 10 times more today than the average for the last week). To do this effectively would again require more sophisticated data logging and analysis.

B. Collecting Data

To collect data from an application running in production, we need a way of continuously profiling, without introducing an unacceptable overhead to the performance of the application. Several tools and approaches exist that may give us fruitful starting points. Many popular APM tools, for example NewRelic[1], allow profiling agents to be installed on production machines to monitor key performance indicators and ship this data to a central analysis system. This is very similar to what we want to do, but the level of profiling information that we require – for example which branch of a conditional is taken – in order to provide deep insights to the developer may be rather more fine-grained than these off the shelf products provide.

In order to make the toolset useful across a large organisation, it needs to be easy to install for every service. Google introduced an organisation-wide tool for profiling that is part of their template build for all production services, meaning that every system can be profiled, without the developers specifically having to consider this as a requirement [5]. Opsian [6] provides an infrastructure for *continuous profiling*, which works via agents that monitor the JVM. The engineering team at Uber recently developed the PyFlame [7] profiler as a high-performance, low overhead profiler for Python applications, which they run in production to collect data to help detecting performance bottlenecks. We plan to investigate these tools further to see if they can be harnesses or adapted for our needs.

V. Research Road Map

We plan to work on ideas, techniques and tooling to address the above problems over the coming months and years. The planned phases of work are as follows:

Our first goal is to build an end-to-end toolchain for collecting data from a web application, aggregating it in a metrics-server, and using this as a source of data for an IDE plugin (likely targetting the JetBrains family of IDEs), rendering the data over the top of the code in a similar manner to the tools of Beck and Cito. This stage will also investigate collecting metrics in a way that does not introduce a significant overhead to the performance of the production application. The plan is to decouple the source of metrics data and analysis from the visualisation as much as possible, to make it easier to build plugins for different IDEs and editors showing the same data.

An area of particular interest within the realm of continuous software engineering is looking not just at snapshots of execution behaviour, but at deltas – particularly linking with the information stored in version control and build systems. With each change, a developer will have a particular intended effect. Can we use this type of analysis and tooling to gather and present data about *changes* in system behaviour, either intended or unintended?

Once this infrastructure is in place, we aim to collect and present data around execution traces, in particular identifying the proportion of executions following different branches. This will require fine-grained instrumentation and profiling.

As we are building tools for developer productivity, our planned evaluation approach centres around developer experience. Does this information help to guide developers' work, and allow them to evaluate the impact of their changes as they continuously update different parts of the system? Is it more meaningful to calculate statistics independently for every newly deployed version of the system? Or is it more useful to present aggregates over all versions since the last time that a particular piece of code was edited? Does presenting these statistics prove a useful source of information to the developer in detecting and correcting problems? Does it inform the way that they work? Or does it simply prove a distraction?

To assess whether or not our prototype tools are really effective in practice is difficult in this area of work, as we need real systems and representative production data in order to make them work. While we could simulate this, impactful research in this area will depend on fruitful collaboration between researchers and practitioners. We hope that we will be able to foster relationships with colleagues in both halves of the software engineering world in order to advance the state of the art in this area.

[1]https://newrelic.com/products/application-monitoring

References

[1] F. Beck, O. Moseler, S. Diehl, and G. Rey, "In Situ Understanding of Performance Bottlenecks through Visually Augmented Code," in *2013 IEEE 21st International Conference on Program Comprehension (ICPC)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2013, pp. 63–72. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICPC.2013.6613834

[2] F. Beck, H. Siddiqui, A. Bergel, and D. Weiskopf, "Method Execution Reports: Generating Text and Visualization to Describe Program Behavior," in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2017, pp. 1–10. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/VISSOFT.2017.11

[3] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth, "Runtime Metric Meets Developer - Building Better Cloud Applications using Feedback," *PeerJ PrePrints*, vol. 3, p. e985v1, Apr. 2015. [Online]. Available: https://doi.org/10.7287/peerj.preprints.985v1

[4] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall, "PerformanceHat: Augmenting Source Code with Runtime Performance Traces in the IDE," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183481

[5] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data centers," *IEEE Micro*, pp. 65–79, 2010. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68

[6] S. Jaffer and R. Warburton, "What is Continuous Profiling?" February 2019, [Online; posted 07 February 2019]. [Online]. Available: https://www.opsian.com/blog/what-is-continuous-profiling/

[7] E. Klitzke, "Pyflame: Uber Engineering's Ptracing Profiler for Python," September 2016, [Online; posted 27 September 2016]. [Online]. Available: https://eng.uber.com/pyflame/