A Case Study on Runtime Verification of Continuous Deployment Process

Shoma Ansai[1] and Masaki Waga[1,2]

[1] Graduate School of Informatics, Kyoto University, Kyoto, Japan
[2] National Institute of Informatics, Tokyo, Japan

Abstract. We report our experience in applying runtime monitoring for a continuous deployment process. Many modern web applications are automatically deployed via continuous deployment (CD). If a deployment process is not successful, an old version of the software will be kept in use, and, for instance, a vulnerability may remain in the running software. To confirm that a new version of software is successfully deployed, we applied runtime monitoring to a continuous deployment process. Our target system consists of GitHub Actions, GitHub Container Registry (GHCR), FluxCD, and an application running on Kubernetes. Through the case study, we find that the deployment action does not always occur within five minutes after image creation, whereas it always occurs within ten minutes. Moreover, our results show that the monitoring tool SyMon we used is efficient enough to monitor the CD process in real time.

Keywords: Runtime verification · Symbolic monitoring · Continuous deployment · Image deployment

1 Introduction

Modern web applications are usually deployed using containers. A container is a lightweight virtual environment. By deploying an application along with a container, it enables seamless server setup, as the container includes all required libraries, runtime environments, and configurations. Application developers build an image and then deploy it to production environments in the form of a container.

At the same time, Continuous Deployment (CD) is gaining attention. CD is a software development strategy in which code changes are automatically deployed to the production environment. It enables rapid and safe deployment. CD can be divided into the following two processes.

1. **Building Phase**: Build an image from the source code.
2. **Deployment Phase**: Create a container based on the built image.

There are three main ways in which the second step can be triggered after the first step is completed.

– Manually start the deployment.

– Trigger the deployment immediately after the image build is completed.
– Periodically poll the container image registry, and deploy the image when a new version is found.

Tools such as FluxCD [2] and ArgoCD [1], which run in the Kubernetes [9] environment, operate based on the third approach. However, this method presents a challenge. Because the deployment process is triggered asynchronously after the image is built, it becomes important to confirm that the new image has actually been deployed. Failure to deploy the new image can cause serious problems. For example, if a security patch is built into a new image but the image is not deployed due to an unforeseen error, the vulnerability may remain in production for an extended period, potentially leading to critical security breaches.

In this paper, we report our experience in applying runtime verification for CD processes to detect unexpected behavior. Specifically, we monitored the log of a CD process based on FluxCD using a monitoring tool, SyMon [16]. The monitoring results show that the deployment action does not always occur within five minutes after image creation, whereas it always occurs within ten minutes. Moreover, our results show that SyMon is efficient enough to monitor the CD process in real time.

2 Related work

Continuous Integration (CI) mechanism is widely used to verify deployed applications [13,14,17], but to the best of our knowledge, there are no examples of verification of the CD system itself.

It is common practice to use application metrics to monitor whether the system is running properly [7,8,10,15,18]. However, we have not found any examples of confirming that the CD system is functioning properly by monitoring whether the output logs meet the expected time constraints.

A method for statically verifying Kubernetes behavior has been proposed [12]. However, this method only verifies the behavior of deploying Kubernetes pods and cannot be used to verify the behavior of the entire CD system. A mechanism for monitoring Kubernetes events in real time and detecting abnormal behavior has also been proposed [11], but no examples of monitoring whether the system meets specific time constraints have been found.

3 Symbolic monitoring with SyMon

SyMon [16] is a tool for symbolic monitoring of *timed data words* against *parametric timed data automata (PTDAs)*. A timed data word is a *timed word* [3] equipped with infinite domain data (e.g., strings and numbers) that represent, for instance, identifiers or sensed values. Specifically, a timed data word is a sequence of (finite domain) events associated with infinite domain data and timestamps. Listing 1 shows a concrete timed data word that SyMon can handle, where the set of events is $\{\mathrm{create}, \mathrm{fetch}\}$. Each line in Listing 1 represents

```
create auth-backend stg-7c03f5241c93d6e77bb132d8ea9ffe9e59e7b62d-1445 171982
fetch auth-example stg-379cca639565f93fe2485c6f443b1d5b45285534-1441 172084
fetch auth-example stg-379cca639565f93fe2485c6f443b1d5b45285534-1441 172085
create auth-frontend stg-7c03f5241c93d6e77bb132d8ea9ffe9e59e7b62d-1445 172140
fetch auth-frontend stg-7c03f5241c93d6e77bb132d8ea9ffe9e59e7b62d-1445 172146
```

Listing 1: A log compatible with SyMon. Each line represents an event equipped with infinite domain data and a timestamp. The first column shows the event, the second column shows the package name, the third column shows the package tag, and the fourth column shows the timestamp.

an event associated with two string values representing the package name and package tag (second and third columns), and a timestamp (fourth column).

PTDAs are a generalization of *parametric timed automata* [4] (which extend timed automata [3] with parameters in timing constraints) to handle infinite domain data. Informally, a PTDA is an NFA equipped with *clock variables* and *data variables* to represent constraints on the time gap between events and the infinite domain data on events, respectively. More specifically, each transition of a PTDA is labeled with parametric constraints and updates on clock and data variables. See [16] for the details of PTDAs.

The semantics of PTDAs is defined with respect to valuations of the timing and data parameters in the constraints. Namely, the parametric constraints in a PTDA are instantiated with the parameter valuations, and its language is defined based on the instantiated concrete constraints. Given a timed data word w and a PTDA $\mathcal{A}$, SyMon returns the prefixes of w accepted by $\mathcal{A}$ along with the corresponding set of parameter valuations η. Thanks to the parameters, one can formulate a PTDA $\mathcal{A}$ with unknown values, and SyMon can detect the acceptance of prefixes of w along with the concrete instance of the unknown values. For example, one can represent a violation of certain requirements *for some* identifier of interest as a PTDA and use SyMon to detect its violation and obtain the concrete identifier for each detected violation.

In addition to PTDAs, SyMon supports a high-level specification language defined based on *timed regular expressions* [6]. Listing 2 shows an example. In this expression, two kinds of events (“`create`” and “`fetch`”) and two parameters over strings (“`current_name`” and “`current_tag`”) are used. To concisely define the main expression, four subexpressions (“`ignore_any`”, “`ignore_irrelevant`”, “`correct`”, and “`failed`”) are defined. SyMon constructs a PTDA from a high-level expression and performs monitoring using it. In our case study, we only use this high-level expression rather than PTDAs. The details of the high-level expression are omitted.

```
#!/usr/local/bin/symon -dnf
var {
    current_name: string;
    current_tag: string;
}
signature create {
    name: string;
    tag: string;
}
signature fetch {
    name: string;
    tag: string;
}
expr ignore_any {
    zero_or_more {
        one_of {
            create(name, tag)
        } or {
            fetch(name, tag)
        }
    }
}
expr ignore_irrelevant {
    zero_or_more {
        one_of {
            create(name, tag | name != current_name || tag != current_tag)
        } or {
            fetch(name, tag | name != current_name || tag != current_tag)
        }
    }
}
expr failed {
    create(name, tag | name == current_name && tag == current_tag);
    within (>300) {
        zero_or_more {
            one_of {
                ignore_irrelevant
            } or {
                create(name, tag | name == current_name && tag == current_tag
)
            }
        };
        one_of {
            create(name, tag)
        } or {
            fetch(name, tag)
        }
    }
}
ignore_any;
failed
```

Listing 2: SyMon's specification

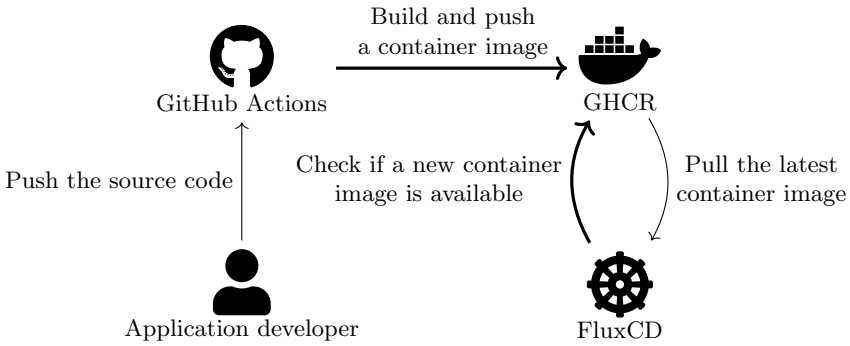

Fig. 1: Outline of the target system. Events observed by the monitor correspond to the actions shown by thick arrows.

4 Case study

4.1 Target system

We monitored a CD system shown in Fig. 1. The system consists of GitHub Actions, GitHub Container Registry (GHCR), FluxCD, and an application running on Kubernetes. Application developers first push the application's source code to GitHub. GitHub Actions then automatically starts, builds an image based on the source code, and pushes that to GHCR. FluxCD regularly polls the Image Registry, and when it finds a new image, it updates the image of the application running on Kubernetes.

The logs output from the CD system components were used to confirm that the expected time constraints were met. Two types of logs were collected: the first is the Building Phase log, which indicates that the image was pushed to GHCR; the second is the Deployment Phase log, which indicates that FluxCD accesses GHCR to see if the image is updated.

We collected the first log as follows. When an image is pushed to GHCR, GitHub's webhook feature notifies an external API server of the event. We created a Registry Monitor to receive the webhooks. When receiving a webhook request, it prints to standard output a log of the images that were pushed. The log is output in JSON format and includes the package name and tag name as shown in Listing 3.

```
{
  "time":"2025-07-02T02:50:46.42462649Z",
  "package_name":"auth-frontend",
  "package_tag":
    "stg-9c8f5e28c2c7d78da2648f5eaa62216038cbd1fd-1458"
  ....
}
```

Listing 3: Log indicating that an image has been pushed

The second log is obtained from the FluxCD component: the FluxCD system periodically polls the image registry and outputs the information of the latest images. The log is output in JSON format as shown in Listing 4, with the package and tag names included in the msg field statement.

```
{
  "level":"info",
  "ts":"2025-07-03T07:06:59.990Z",
  "msg":"Latest image tag for
   ghcr.io/piny940/auth-frontend resolved to
   stg-9c8f5e28c2c7d78da2648f5eaa62216038cbd1fd-1458...",
...
}
```

Listing 4: Log of FluxCD polling GHCR

We preprocessed these logs and converted them to a format compatible with SyMon. The logs after pre-processing are shown in Listing 1. Specifically, we performed the following three pre-processing steps.

1. write labels indicating the type of logs: “create” for logs indicating that an Image has been pushed, and “fetch” for logs of FluxCD polling GHCR
2. extract package name and tag name from the log
3. convert timestamp to UNIX time

4.2 Specification

We tested whether FluxCD detects image updates within five minutes and within ten minutes after they are pushed to GHCR. We used SyMon’s specification shown in Listing 2.

Lines 2 through 5 of the specification declare variables for specifying the name and tag to be monitored. Only events that match these values are focused on. Lines 6 through 13 define create and fetch as log events. The “`ignore_any`” defined in lines 14 through 22 means that any number of “`create`” or “`fetch`” events can occur. This is written to ignore harmless parts of the event sequence. Lines 23 to 31 define “`ignore_irrelevant`”, which is written to ignore “`create`” or “`fetch`” events that do not match “`current_name`” or “`current_tag`” as ‘irrelevant events’. Lines 32 to 48 define “`failed`”, which is the main body of this specification, i.e., the anomaly to be detected. The content is represented in the following three steps.

1. A create event occurs(l33).
2. Only events other than fetch that match name and tag in “`current_name`” and “`current_tag`” occur, and 300 seconds pass(l35-l41).
3. Some event occurs(l42-l46).

Lines 49 and 50 state, ‘Detect if “`failed`” occurs after “`ignore_any`”.’

Table 1: Time taken to execute SyMon

Number of Days	Number of Entries	Execution Time(ms)
5	12758	360
10	25223	376
15	41151	398

4.3 Results and discussions

We checked if the system's behavior met the specification mentioned in Section 4.2. We used five-day logs. There were 12758 log entries and 12 "`create`" logs were included. Logs used for benchmarking are available on GitHub [5].

When tested with the specification "within five minutes", we found five logs that did not meet the constraint, as shown in Listing 5. This indicates that the deployment was not always done within five minutes after the image was pushed. It should be noted that multiple "`fetch`" logs were detected for the same "`create`" log. On the other hand, when tested with the specification "within ten minutes", all logs met the constraint. This means that the deployment always occurred within ten minutes.

```
@1751425023.000000.     (time-point 9443)       x0 == auth-frontend     x1 ==
      stg-9c8f5e28c2c7d78da2648f5eaa62216038cbd1fd-1458 true
@1751425023.000000.     (time-point 9443)       x0 == auth-example      x1 ==
      stg-9c8f5e28c2c7d78da2648f5eaa62216038cbd1fd-1458 true
@1751425050.000000.     (time-point 9444)       x0 == auth-example      x1 ==
      stg-9c8f5e28c2c7d78da2648f5eaa62216038cbd1fd-1458 true
@1751425050.000000.     (time-point 9444)       x0 == auth-frontend     x1 ==
      stg-9c8f5e28c2c7d78da2648f5eaa62216038cbd1fd-1458 true
@1751425052.000000.     (time-point 9445)       x0 == auth-example      x1 ==
      stg-9c8f5e28c2c7d78da2648f5eaa62216038cbd1fd-1458 true
```

Listing 5: output of SyMon

We also measured the time it takes to run SyMon against 5, 10, and 15 days of logs and evaluated the time it takes to run the test. The time taken to execute SyMon is shown in Table 1. It took 360 milliseconds to check 12758 logs for five days. For comparison, we ran a test on 25223 log entries for ten days and 41151 log entries for 15 days, and it finished in 376 and 398 milliseconds, respectively. This indicates that execution time remains within a realistic time frame even as the number of logs increases.

5 Conclusions and perspectives

In this paper, we used SyMon to confirm that the CD system operates according to the specifications. The event logs showed that the deployment was not always done within five minutes after image creation. On the other hand, it always occurred within ten minutes.

In terms of areas for improvement in SyMon, one issue is that when JSON-formatted logs are provided, it should be possible to verify the values of each

field without preprocessing. Currently, SyMon cannot handle JSON-formatted logs directly, so it is necessary to preprocess the logs into a format that SyMon can handle. However, describing preprocessing individually for each system is a significant burden in terms of implementation. We would like to further improve SyMon to create a practical monitoring system.

Acknowledgements This work is partially supported by JSPS KAKENHI Grant No. 22K17873, JST BOOST Grant No. JPMJBY24H8, and JST PRESTO Grant No. JPMJPR22CA.

References

1. Argo CD: Declarative Continuous Delivery for Kubernetes (2025), https://argo-cd.readthedocs.io
2. Flux: Open and Extensible Continuous Delivery for Kubernetes (2025), https://fluxcd.io
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
4. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: STOC. pp. 592–601. ACM (1993)
5. Ansai, S., Waga, M.: Symon (2025), https://github.com/piny940/SyMon/tree/master/example/final
6. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM **49**(2), 172–206 (2002)
7. Boniol, P., Liu, Q., Huang, M., Palpanas, T., Paparrizos, J.: Dive into time-series anomaly detection: A decade review (2024), https://arxiv.org/abs/2412.20512
8. Brejcak, P.: Real-world insights: Anomaly detection in internet traffic (2024), https://tech.trivago.com/post/2024-02-13-real-world-insights-anomaly-detection-in-internet-traffic
9. Burns, B., Grant, B., Oppenheimer, D., Brewer, E.A., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (2016)
10. Katz, S., Ramachandran, J., Butsch, J., Lau, P., Vaithilingam, R., Burrell, G.: Telltale: Netflix application monitoring simplified (2020), https://netflixtechblog.com/telltale-netflix-application-monitoring-simplified-5c08bfa780ba
11. Kumar, T.V.: Enhanced kubernetes monitoring through distributed event processing (1st edition). International Journal of Research in Electronics and Computer Engineering **12**(3), 1–16 (2024)
12. Liu, B., Lim, G., Beckett, R., Godfrey, P.B.: Kivi: Verification for cluster management. In: 2024 USENIX Annual Technical Conference (USENIX ATC 24). pp. 509–527. USENIX Association, Santa Clara, CA (Jul 2024), https://www.usenix.org/conference/atc24/presentation/liu-bingzhe
13. Nadeem Ahmad, R.S.: Improving pull request confidence for the netflix tv app (2021), https://netflixtechblog.medium.com/improving-pull-request-confidence-for-the-netflix-tv-app-b85edb05eb65
14. Shahin, M., Ali Babar, M., Zhu, L.: Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access **5**, 3909–3943 (2017). https://doi.org/10.1109/ACCESS.2017.2685629

15. Srivatsan, S.: Observability at scale: Building uber's alerting ecosystem (2018), https://www.uber.com/en-JP/blog/observability-at-scale/
16. Waga, M., André, É., Hasuo, I.: Symbolic monitoring against specifications parametric in time and data. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 520–539. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_30, https://doi.org/10.1007/978-3-030-25540-4_30
17. Xiaoyang Tan, Y.L., Balabanov, S.: Flaky tests overhaul at uber (2024), https://www.uber.com/en-JP/blog/flaky-tests-overhaul/
18. Zhang, Z., Nie, K., Yuan, T.T.: Moving metric detection and alerting system at ebay (2022), https://arxiv.org/abs/2004.02360