2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)

DeployFix: Dynamic Repair of Software Deployment Failures via Constraint Solving

Haoyu Liao[1] Jianmei Guo[1,†] Bo Huang[1] Yujie Han[1] Dingyu Yang[2] Kai Shi[3]
Jonathan Ding[4] Guoyao Xu[3] Guodong Yang[3] Liping Zhang[3]
[1]*East China Normal University* [2]*Zhejiang University* [3]*Alibaba Group* [4]*Intel*

Abstract

Software deployment misconfiguration often happens and has been one of the major causes of deployment failures that give rise to service interruptions. However, there is currently no existing approach to automatically repairing deployment failures. We propose DeployFix, which automatically repairs software deployment failures via constraint solving in the dynamic-changing deployment environments. DeployFix first defines DeployIR as a unified intermediate representation to achieve the translation of heterogeneous specifications from different schedulers with different syntaxes. By reducing the root-cause analysis of deployment failures to the conflict resolution in propositional logic, DeployFix uses off-the-shelf constraint solvers to achieve automatic localization and diagnosis of conflicting constraints, which are the root causes of deployment failures. DeployFix finally resolves the conflicting constraints and generates repaired deployment configurations in terms of practical requirements. We evaluate DeployFix in both simulation and production environments with tens of thousands of nodes at Alibaba, on which tens of thousands of applications are running guided by hundreds of thousands of deployment constraints. Experimental results demonstrate that DeployFix outperforms the state of the art and it correctly repairs the deployment failures in minutes, even in a large production data center.

Keywords

Cloud Computing, Deployment Failures, Dynamic Repair, Constraint Solving

ACM Reference Format:
Haoyu Liao, Jianmei Guo and Bo Huang, Yujie Han, Dingyu Yang, Kai Shi, Jonathan Ding, Guoyao Xu, Guodong Yang and Liping Zhang. 2024. DeployFix: Dynamic Repair of Software Deployment Failures via Constraint Solving. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695268

† Corresponding Author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695268

1 Introduction

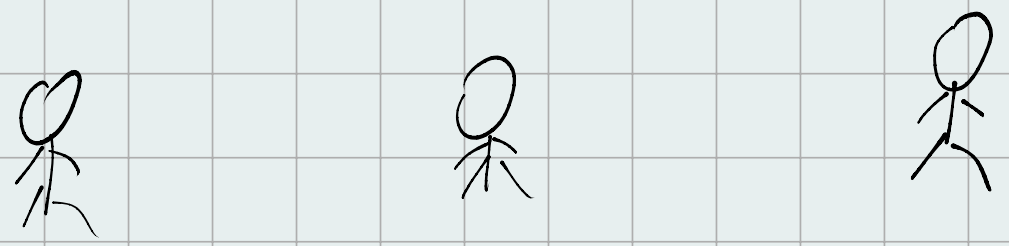
It is practically impossible and error-prone to manually deploy various applications, such as e-commerce, databases, big data processing, and large models to many cloud instances in a large data center. Instead, cloud engineers code deployment configurations into cloud schedulers (e.g., write YAML/JSON files in Kubernetes) to meet the requirements of both cloud consumers and providers. The deployment configurations essentially serve as *deployment constraints* and help cloud schedulers assign applications to appropriate nodes.[2] However, misconfiguration often happens and has been one of the major causes of *deployment failures* [1–9] that give rise to service interruptions. Even a completely bug-free program could behave incorrectly due to misconfiguration. Fast repairing misconfigurations is crucial to the failure recovery and reducing financial and reputational losses.

Most existing techniques known as *static repair* input buggy programs or misconfigurations and output fixed ones. Those techniques include automatic program repair (APR) [10–19], configuration bug detection [20–34] and software product line reconfiguration [11, 12, 35–37]. However, even a correct configuration could become buggy due to the dynamic changes in the deployment environments, for example, due to the frequent scheduling and rolling updates of the applications and nodes. In contrast, *dynamic repair* must first determine where and when a misconfiguration happens in the dynamically-changing environments and then repair it without triggering cascading failures.

Given the public reports from large cloud service providers [38–40] and Kubernetes [41], currently, no approach exists for automatically repairing deployment failures or misconfigured schedulers. Cloud engineers mainly rely on their experience and manual inspection to troubleshoot and repair deployment failures. For example, according to customer requirements, cloud engineers encode three deployment constraints in different applications' YAML files for Kubernetes, that is, constraints `C1: app1 requires app2`, `C2: app1 requires app3`, and `C3: app2 excludes app3`. These *affinity* (`requires`) or *anti-affinity* (`excludes`) constraints are the most common implementations of deployment constraints to maximize resource utilization and minimize resource fragmentation while guaranteeing the quality of service (QoS) [42–44]. At first, engineers did not notice any problem. Once deployment failures happen, engineers have to manually inspect each YAML file and check out the conflicts among the constraints. Technically, there are three challenges for engineers to inspect the misconfigurations and repair them automatically.

[2]In the following context, a node refers to a physical machine, virtual machine or container used to run the cloud instances of applications.

Kubernetes 架构



Load Balancer

实例1　实例2　实例3

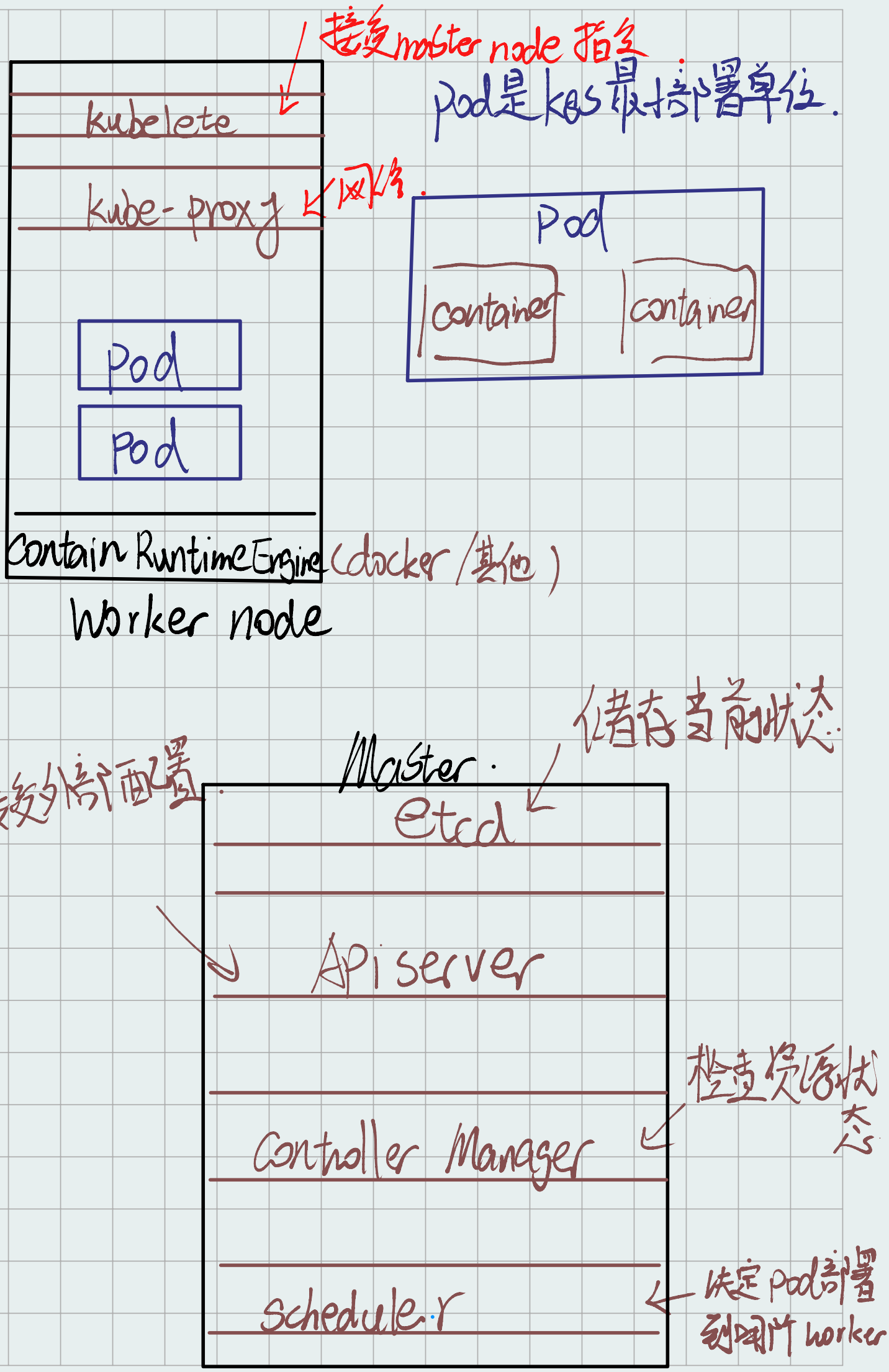
Master node

掌握集群状态
部署创造的 worker

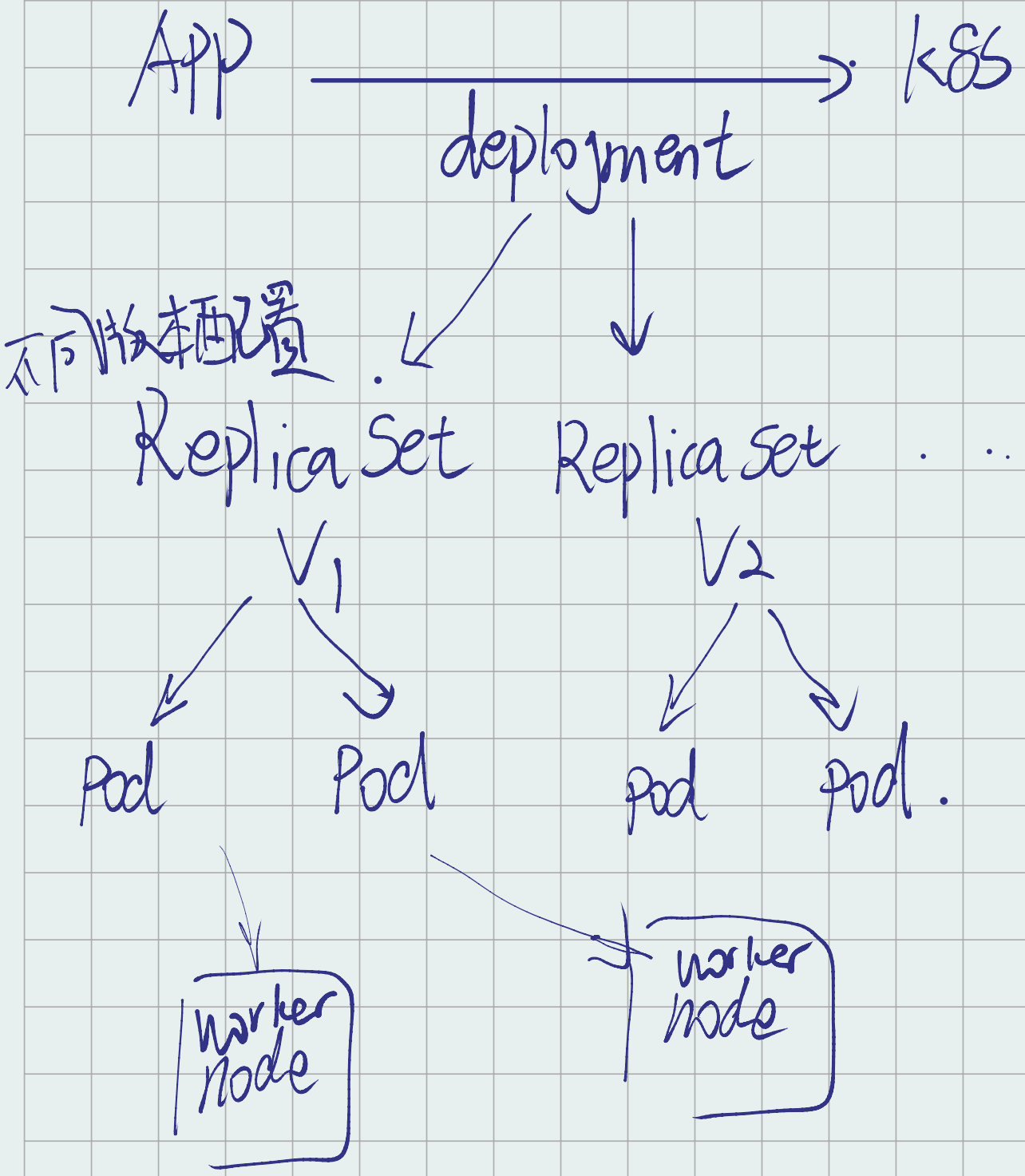
worker　负责运行 container

worker node

一台机器 / vm.


接受master node指令
kubelete
kube-proxy
网络
Pod
Pod
Pod
container
container
Contain Runtime Engine (docker/其他)
Worker node
Pod是k8s最小部署单位.
储存当前状态
接受外部配置
Master
etcd
Apiserver
controller Manager
检查资源状态
schedule.r
决定pod部署到哪个worker

Pod的部署由deployment来完成



(1) Heterogeneous Specification: Different schedulers support multiple heterogeneous syntaxes to express deployment constraints. For example, Kubernetes and Apache Yarn have obvious differences in their syntaxes. Even for the same scheduler, different specifications exist, e.g. `nodeAffinity`, `podAffinity` and `podAntiAffinity` in Kubernetes. Furthermore, multiple operators are supported to enrich the semantics, e.g. `In`, `NotIn`, `Exists`, and `DoesNotExist` in `podAffinity`.

(2) Complex Root-Cause Analysis: In the example above, when adding constraint `C3` to `app2`'s configurations, the deployment of `app1` fails. However, there is no modification in `app1`'s configurations. In practice, there are tens of thousands of applications deployed in a data center, and the deployment constraints among them are complex, probably leading to the complicated *cascading effects*. This hides the root causes of the deployment failures, demanding rigorous logical reasoning to achieve troubleshooting.

(3) Automatic Repair: Real-world failures can be complicated, and there could be multiple feasible solutions for different conditions. Different repair strategies could lead to different results, and some of them could not be applied to all conditions or make engineers confused. Still using the example above, removing all three constraints `C1` ~ `C3` or removing any of them are feasible. Replacing the *requires* in `C1` and `C2` with *excludes* is also feasible. However, a feasible solution may not be the optimal. The finally-deployed solution is usually subject to the practical requirements and engineering decisions, e.g., guaranteeing the QoS of critical applications. Though the solution generated is not the optimal, automatic repair of deployment failures facilitates minimizing the service downtime and reducing the losses.

To address these challenges, we propose DeployFix, an approach that automatically troubleshoots and repairs deployment failures based on the current deployment environments. First, to support heterogeneous specifications, we define a unified intermediate representation for different deployment constraints, called DeployIR, to shield the syntax differences of schedulers. With DeployIR, DeployFix translates the deployment configurations of running applications and nodes from heterogeneous schedulers into a unified representation.

Second, we observe that the affinity and anti-affinity constraints can essentially be expressed by the *requires* and *excludes* semantics. To achieve automatic localization and diagnosis, we reduce the root-cause analysis of deployment failures to the conflict resolution in propositional logic. DeployFix encodes the DeployIR translated from running applications and nodes in the dynamic deployment environment into propositional constraints and then feeds the constraints into off-the-shelf constraint solvers. By solving the minimal unsatisfiable core (MUC), DeployFix correctly identifies the conflicting constraints, which are the root causes of deployment failures.

Third, cloud applications are often classified into different priorities [42, 44–46]. Given the production requirement at Alibaba that safeguarding high-priority applications is the first place in the event of a deployment failure, because these applications are often mission critical. To achieve automatic repair, the default repair strategy of DeployFix is to remove the conflicting constraints that block the deployment of critical applications. DeployFix preserves the tracing data including the file path and line number of the deployment constraints in DeployIR to help synthesize the finally repaired configuration file as the automatic solution. In addition, DeployFix allows engineers to change the default repair strategy and select another solution.

To summarize, we have made the following contributions:

(1) We propose DeployFix, an approach that supports the dynamic repair of deployment failures in a large data center with massive and constantly evolving deployment constraints. DeployFix designs DeployIR as a unified representation to achieve translation of heterogeneous syntaxes from running applications and nodes. By reducing the root-cause analysis of deployment failures to the conflict resolution in propositional logic, DeployFix uses off-the-shelf constraint solvers to achieve automatic troubleshooting of conflicting constraints. According to the repair strategy and the tracing data, DeployFix generates the finally repaired configuration files that can be directly applied to the scheduler as the automatic solution. To the best of our knowledge, we are the first to propose a full, scalable solution to dynamic repair that accurately identifies different configuration syntaxes, efficiently detects and diagnoses the root causes, and automatically generates fixes, for various, potentially changing, deployment failures in data centers.
(2) To evaluate the correctness and efficiency of DeployFix, we conducted a simulation experiment with a small set of manually verified deployment configurations and a large set of randomly-generated deployment configurations to check whether DeployFix can automatically troubleshoot and repair the conflicting constraints. The results show that DeployFix can correctly identify the root causes within minutes and generate a feasible solution.
(3) To further demonstrate the practicality and scalability, we deployed DeployFix in a production data center with tens of thousands of nodes and applications and hundreds of thousands of deployment constraints at Alibaba, one of the largest e-commerce providers in China. The deployment constraints involve both the affinities and anti-affinities among nodes and applications. According to the replay and comparison of past failure reports caused by misconfigured deployment constraints, DeployFix automatically repairs the failures within minutes, while engineers may take hours to manually troubleshoot and repair the failures.
(4) We implement DeployFix in 5,000 lines of Rust with the Z3 Solver [47], and the source code is publicly available at https://github.com/solecnugit/deployfix to facilitate reproducible research and further study.

2 Motivation

2.1 Motivating Example

We present a motivating example in Figure 1 to illustrate the importance and difficulty of guaranteeing the satisfiability of deployment constraints in a large data center. First, `appA` has a dependency requirement for `appB` for reducing the network communication latency. Based on the requirement, the cloud engineers added a new constraint `C1: appA requires appB`, as illustrated in Figure 1(a), which enables during scheduling.

appA.yaml

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: app
              operator: In
              values:
                - appB
        topologyKey: kubernetes.io/hostname
```

(a) Part of deployment configuration of `appA` indicating that it should be co-located with appB on the same set of nodes during scheduling

appA.yaml (Patch)

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: performance
              operator: NotIn
              values:
                - high-performance
```

(b) Part of configuration to patch `appA` indicating that it **should not** be deployed on the node with label `performance=high-performance` during scheduling

appB.yaml (Patch)

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: performance
              operator: In
              values:
                - high-performance
```

(c) Part of configuration to patch `appB` indicating that it **should** be deployed on the node with label `performance=high-performance` scheduling

Scheduling Results

```
NAME      READY  STATUS   ... NODE
appA-1    1/1    Running  ... high-performance-node
appA-2    0/1    Pending  ... <none>
appB-1    1/1    Running  ... normal-node
appB-2    1/1    Running  ... normal-node
```

(d) Part of scheduling result when scaling up `appA`. `appA-2` can never be `Running` since there is no `high-performance-node` with an instance of `appB`. The default parameters of Kubernetes do scale up one by one.

Figure 1: A motivating example of a deployment failure caused by conflicting deployment constraints.

Second, as shown in Figure 1(b), according to the performance characteristics of `appA`, `appA`'s developers thought it was a better choice to forcibly ban `appA` from `high-performance nodes (HPNs)` to reduce cost. The engineers added a constraint `C2: appA excludes HPN`, which enables during scheduling, to `appA`'s configurations and all the instances of `appA` were evicted from `HPNs`. Because there were some running instances of `appB` on `normal nodes`, the deployment of `appA` on `normal nodes` was not affected.

Third, to better guarantee performance, `appB`'s developers told the engineers that `appB` should be deployed on `HPNs`. The cloud engineers added a constraint `C3: appB requires HPN` enabling during scheduling and launched a rolling update to deschedule all the instances of `appB` to `HPNs`. Because constraint `C2` enables only during scheduling, all the instances of `appA` ran normally with a disregarded performance downgrade, which refers to Figure 1(c).

Finally, engineers try to launch more instances of `appA` to handle the bursty requests. However, there is no `normal node` with `appB` available now, making it impossible to scale `appA` up. Without enough instances, the performance of `appA` downgrades, causing interruptions, as shown in Figure 1(d).

As aforementioned, two major problems that caused the failure can be summarized as follows: (1) The cascading effect arising from complex application relationships makes the changes in the configurations of the application itself not the only place where the root causes could arise, but also the dynamic changes in the deployment environments. (2) The rich semantics of deployment constraints hide the conflict from being promptly discovered.

2.2 State of the Art

To the best of our knowledge, there is no existing approach to automatic repair of deployment failures at present. Currently, as illustrated in Figure 1(d), cloud engineers typically inspect the deployment configurations when deployment failures happen in practice. When an application cannot be deployed normally, engineers first check the scheduler logs to determine whether the failure is caused by conflicting deployment constraints. If so, engineers check the configurations and recent change histories of the application, which refers to Figure 1(a) and Figure 1(b) in our example. If there are node affinities, engineers check whether there are any spelling mistakes or whether the nodes with such labels exist. If there are node anti-affinities, engineers check whether the corresponding labels exist on all the candidate nodes. If so, the application might be scheduled to the wrong cluster. If there are inter-application affinities, engineers also check whether spelling mistakes exist, such as the configuration shown in Figure 1(c). Furthermore, they confirm whether the target applications of affinities can be deployed normally. If there are inter-application anti-affinities, engineers check whether there are any idle nodes without target applications running on them to determine whether the failure is caused by resource shortages. As for the failures caused by the cascading effect of node affinities and inter-application affinities, engineers recursively inspect all the related deployment configurations, which could be laborious and time-consuming.

2.3 Challenges

To better understand the challenges of repairing deployment constraints in a large data center and bridge the gap between the deployment configurations and deployment constraints, we further conducted a preliminary study within the production environments and empirically analyzed the challenges. According to the interviews with our engineers, who have extensive experience in cloud scheduling, we identified three major challenges in repairing deployment failures caused by conflicting deployment constraints.

2.3.1 Heterogeneous Expressions Different schedulers express deployment constraints in heterogeneous syntaxes. Even in the same scheduler, the specifications of deployment constraints between a node and an application, and between two applications are also heterogeneous. Furthermore, each specification supports multiple operators to enrich the semantics. For example, constraint `C1` in Figure 1(a) can be expressed by either `podAffinity` with the `In` operator or `podAntiAffinity` with the `NotIn` operator and constraint `C2` can be expressed by either `nodeAffinity` or `nodeSelector` in Kubernetes.

2.3.2 Complex Root-Cause Analysis In a large data center, there are tens of thousands of nodes and applications, leading to hundreds of thousands of complex deployment constraints. These constraints are distributed in overwhelming deployment configurations. The rich semantics of the schedulers further complicate the relationship between nodes and applications, forming complicated cascading effects. Using the example in Figure 1, the deployment failure of `appA` is caused by constraint `C3`, which does not belong to `appA`'s configurations. Furthermore, constraint `C2` enables only during scheduling, making the failure probably not immediately happen but until the next time of scheduling `appA`. The delay between the misconfiguration and the deployment failure makes it time-consuming to identify the root causes based on the change history.

2.3.3 Automatic Repair As described in Section II-B, the state-of-the-art requires cloud engineers to manually inspect every related deployment configuration. The requirement for engineers to complete the repair within minutes is beyond human capabilities. However, to reduce the losses, the service downtime has to be minimized, raising an emergency need for an automatic approach. Nevertheless, the feasible solutions may not be unique. Still using the above example, removing all three constraints `C1` ~ `C3` or any of them can be a feasible solution. The finally-deployed solution is usually subject to practical requirements and engineering decisions, such as the priorities of `appA` and `appB`.

3 Methodology

3.1 Framework Overview

The goal of DeployFix is a full solution that accurately identifies different configuration semantics, efficiently detecting and diagnosing the root causes, and automatically generating fixes. Figure 2 illustrates the framework of DeployFix for repairing deployment failures, including four steps: *Translation*, *Formalization*, *Conflict Resolution*, and *Code Generation*.

First, DeployFix translates deployment configurations of running applications and nodes from heterogeneous schedulers into DeployIR. Second, by reducing the root cause analysis of deployment failures to conflict resolution in propositional logic, DeployFix formalizes the DeployIR into the corresponding propositional constraints. Third, DeployFix identifies the conflicting constraints based on the MUC from the off-the-shelf solver [47] and remaps the conflict constraints back to the original deployment configurations based on the tracing data in DeployIR. Finally, DeployFix generates repaired deployment configurations to automatically repair the deployment failure.

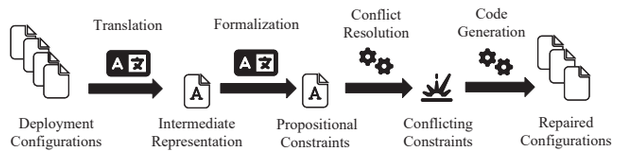


Figure 2: Overview of DeployFix Framework

```
Constraint ::= Source  Operator  Target Separator  TracingData
Source ::= Entity
Target ::= Entity | Entity, Target
Entity ::= Key=Value
Operator ::= requires | excludes
Separator ::= //
TracingData ::= GeneralData | GeneralData;ExtraData
GeneralData ::= Topology;File;Line;Column
Topology ::= Node | Region | Rack
ExtraData ::= DataEntry; | DataEntry; ExtraData
DataEntry ::= Key=Value
```

Figure 3: The Syntax of DeployIR

3.2 Translation

To be compatible with the heterogeneous deployment constraints from different schedulers, we designed the DeployIR. The main purpose of DeployIR is to shield the differences in specifications between schedulers and provide a unified representation. As shown in Figure 3, DeployIR consists of four components: *source entity*, *target entities*, *operator* and *tracing data*. The *source entity* contains one entity, such as an application or a node. The *target entities* can be one or more entities separated by commas. *Operators* reveals the type of the constraint, which is represented by `requires` and `excludes` respectively. *Tracing data* can be divided into two subkinds: *general data* and *extra data*. *General data* stores the general attributes of the constraint, such as the file name, line number, and column number of the original definitions. *Extra data* depends on specific schedulers, e.g. the *kind* in Kubernetes. Tracing data plays an important role in generating the final repaired configuration, helps translate the constraints in the MUC identified by constraint solvers to those in the deployment configurations of different schedulers, and subsequently helps synthesize the final configuration files of schedulers. Figure 4 shows the correspondence between the fields in Kubernetes's configurations and DeployIR, demonstrating how appA's configurations in Figure 1 are translated into DeployIR. In addition, DeployFix supports various topologies, including nodes, regions, and racks. We use nodes as an example in the paper for simplicity and the processing of regions or racks is similar.

To achieve dynamic repair, when translation, DeployFix collects running applications, nodes and enabled deployment constraints, and generates repair solutions always based on the current deployment environments, e.g. before scheduling, after scheduling or rolling updates. If an entity represented in the constraints doesn't exist, e.g. due to a spelling mistake, DeployFix directly reports the conflict and skips the subsequent process with an option enabled.

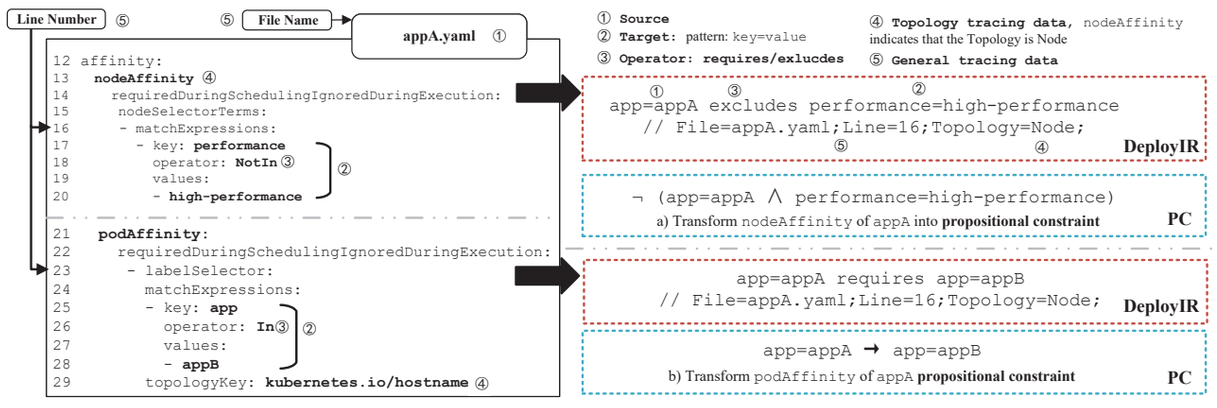


Figure 4: Translating the configurations in Figure 1 into DeployIR and propositional constraints (PC).

3.3 Formalization

3.3.1 Boolean Satisfiability Problem Boolean satisfiability problem (SAT) is a decision problem that determines whether a given Boolean formula can be satisfied [48–51] and the goal of the satisfiability problem is to find a set of assignments of Boolean variables that satisfies the Boolean formula. Modern off-the-shelf solvers can solve millions of variables and constraints in seconds, making it feasible to solve deployment failures on a data center scale through constraint solving [47, 49].

We denote ***solve***(C) searching for an assignment that satisfies the constraints C by the off-the-shelf solver. If there is one assignment that makes C satisfied, ***solve***(C) returns SAT and exactly one satisfying assignment. Otherwise, ***solve***(C) returns $UNSAT$ and the minimal subset of conflict constraints used in the proof of unsatisfiability, called the MUC set. Note that the propositional logic is sufficient to formalize the deployment constraints, so DeployFix does not adopt the first-order logic.

3.3.2 Definitions Let the set of all entities in the data center be $E = \{e_i\} = A \cup N$, where $A = \{a_i\}$ is the set of all applications, $N = \{n_i\}$ is the set of all nodes, $i \in \mathbb{N}$,. We embed the feature labels on nodes. For example, in the motivating example, there are two applications $appA, appB \in A$ and a node with label `performance=high-performance` placed: $n_{\text{HPN}} \in N$. Moreover, for each entity $e \in E$, let C_e be its particular constraints.

Affinity Constraints. Let the affinity constraint between entities e_i and e_j be

$$\forall_{i,j\in\mathbb{N}}, Requires(e_i, e_j) = e_i \rightarrow e_j$$

Thus, all affinity constraints in the data center form

$$C_{\text{aff}} = \bigwedge_{i,j\in\mathbb{N}} Requires(e_i, e_j)$$

Anti-affinity Constraints. Let the anti-affinity constraint between entities e_i and e_j be

$$\forall_{i,j\in\mathbb{N}}, Excludes(e_i, e_j) = \neg(e_i \wedge e_j)$$

Thus, all anti-affinity constraints in the data center form

$$C_{\text{anti}} = \bigwedge_{i,j\in\mathbb{N}} Excludes(e_i, e_j)$$

Deployment Constraints. If the scheduler is willing to deploy an application a_i to a node n_j, the deployment constraints between a_i and n_j can be defined as:

$$C_{a_i,n_j} = C_{a_i} \wedge C_{n_j} \wedge C_{\text{aff}} \wedge C_{\text{anti}}$$

where C_{a_i} is the constraints of application a_i, C_{n_j} is the constraints of node n_j, and C_{aff} and C_{anti} represent the affinity and anti-affinity of all entities in the data center. Note that the deployment constraints can represent the relationships between multiple applications and multiple nodes. For example, if multiple applications $A_k \subset A$ need to be co-located on node n_j, the deployment constraints can be defined as $C_{A_k,n_j} = \bigwedge_{a_i\in A_k} C_{a_i} \wedge C_{n_j} \wedge C_{\text{aff}} \wedge C_{\text{anti}}$.

Given the deployment constraints between application a_i and node n_j, there are two scheduling scenarios: *schedulable scenario* and *unschedulable scenario*.

(1) Schedulable Scenario. A schedulable scenario is defined as

$$\textbf{solve}(C_{a_i,n_j}) \text{ is } SAT$$

indicating that application a_i is schedulable to node n_j while subject to all the deployment constraints between a_i and n_j.

(2) Unschedulable Scenario. An unschedulable scenario is defined as

$$\textbf{solve}(C_{a_i,n_j}) \text{ is } UNSAT$$

indicating that application a_i is unschedulable to node n_j because the deployment constraints between a_i and n_j are unsatisfied,

In the unschedulable scenario, the off-the-shelf solver outputs the MUC set, which identifies the root cause of the unschedulability. Based on the MUC set and the user-defined repair strategy, DeployFix transforms C_{a_i,n_j} into C'_{a_i,n_j}, generating a schedulable scenario, which is a feasible repair solution.

3.3.3 Translation of DeployIR to Propositional Constraints When translating DeployIR into propositional constraints, the rules are summarized as follows: (1) DeployFix translates the affinity and

Table 1: The splitting operations for constraints when entity e_1 has self-anti-affinity or self-affinity constraint, where $e_1, e_2, e_3 \in E$. When both self-affinity and self-anti-affinity exist in e_1, constraints C1 and C4 are conflicting.

Type	Before	After splitting	Id
self-anti-affinity	$Excludes(e_1, e_1)$	$Excludes(e_1', e_1'') \wedge Excludes(e_1'', e_1')$	C1
	$Requires(e_2, e_1)$	$Requires(e_2, e_1') \vee Requires(e_2, e_1'')$	C2
	$Excludes(e_3, e_1)$	$Excludes(e_3, e_1') \wedge Excludes(e_3, e_1'')$	C3
self-affinity	$Requires(e_1, e_1)$	$Requires(e_1', e_1'') \wedge Requires(e_1'', e_1')$	C4

anti-affinity constraints with a single *target entity* into the corresponding *requires* and *excludes* semantics. (2) DeployFix translates affinity constraints with multiple *target entities* into the disjunction of multiple *requires* constraints. (3) DeployFix translates anti-affinity constraints with multiple *target entities* into the conjunction of multiple *excludes* constraints. Figure 4 demonstrates how DeployFix encodes the DeployIR in Figure 1 into propositional constraints. (4) DeployFix handles the corner cases of *self-anti-affinity* specially since a variable can be either *true* or *false*.

Self-anti-affinity constraint is commonly used to limit at most one instance of the application per node. In contrast, self-affinity is less common and could be used to deploy multiple identical instances of the same application to the same node. Table 1 shows how DeployFix splits the constraints when entity e_1 has self-anti-affinity or self-affinity constraints. Note that when an e_1 has both self-affinity and self-anti-affinity, there is a conflict, e.g. C1 and C4 in the table.

3.3.4 Properties To meet the requirements aforementioned, we expect DeployFix to satisfy the following properties.

- *Correctness*. If the deployment constraint between application a_i and node n_j is a schedulable scenario, deploying a_i to n_j should satisfy all the deployment constraints between a_i and n_j.
- *Efficiency*. DeployFix should be able to identify conflicting constraints among a large number of applications and deployment configurations in a short time.
- *Scalability*. DeployFix should handle diverse deployment failures caused by conflicting constraints in practice.

The off-the-shelf constraint solvers, such as Z3, guarantee that, in an unschedualable scenario, they always return *UNSAT* and completely identify the MUC set [47]. Benefiting from this, DeployFix naturally holds the *correctness* property. We verify the remaining two properties in the evaluation section by addressing three research questions.

3.4 Conflict Resolution

3.4.1 Resolving Conflicting Constraints DeployFix gets the MUC set from the solver in an unschedulable scenario, identifies the conflicting constraints by mapping the MUC set to the definition of the constraints with tracing data in DeployIR and predefined tracking variables when feeding deployment constraints to the solver.

Figure 5 shows the process of solving the scheduling scenario of deploying appA and appB on a high-performance node n_{HPN}. DeployFix first feeds $C_{a_{\text{appA}}, n_{\text{HPN}}}$ or $C_{a_{\text{appB}}, n_{\text{HPN}}}$ into the solver, where Figure 5(a) is an unschedualable scenario and the MUC set

```
c0 appA
c1 ⋀(¬ (app=appA ⋀ performance=high-performance))
c2 ⋀(app=appA → app=appB)
c3 ⋀(app=appB → performance=high-performance)
```

UNSAT

(a) Formula of appA on a high-performance node

```
c0 appB
c1 ⋀(¬ (app=appA ⋀ performance=high-performance))
c2 ⋀(app=appA → app=appB)
c3 ⋀(app=appB → performance=high-performance)
```

SAT

(b) Formula of appB on a high-performance node

Figure 5: The formulas of appA and appB in Figure 1 on a high-performance node n_{HPN}. (a) is an unschedualable scenario. The MUC set of appA is constraints C1 ~ C3; (b) is a schedualable scenario.

is C1 ~ C3. Removing any of C1 ~ C3 transforms the $C_{a_{\text{appA}}, n_{\text{HPN}}}$ into $C'_{a_{\text{appA}}, n_{\text{HPN}}}$, a schedualable scenario. On the contrary, Figure 5(b) is a schedulable scenario. A feasible assignment is: constraints C0 and C3 are *true*, and C1 and C2 are *false*.

3.4.2 Resolving Circular Dependencies A schedulable scenario reveals that the scheduler can deploy the application to the node, but how to deploy still depends on the implementation of the scheduler. For example, constraints app1 `requires` app2 and app2 `requires` app1 are two constraints with a circular dependency. Neither of the two applications can be scheduled during a cold start, However, the two formulas: app1 ∧ (app1 → app2) ∧ (app2 → app1) and app2 ∧ (app1 → app2) ∧ (app2 → app1), are both satisfiable. DeployFix supports detecting circular dependencies before solving deployment constraints with the off-the-shelf solver to accelerate the conflict resolution process. DeployFix uses a modified version of Johnson's algorithm [52] to find all the cycles in the directed graph composed of all affinity constraints to identify the circular dependencies. If all the *target entities* of an affinity constraint are blocked by a cycle, then the constraint is a circular dependency. DeployFix will report the constraint as a conflict.

3.5 Code Generation

To minimize downtime and reduce the cost of human intervention, DeployFix generates repaired deployment configuration files to be directly applied by the scheduler based on the conflicting constraints from the "Conflict Resolution" step, default or user-defined repair strategies, and tracing data stored in DeployIR. The step involves two sub-steps: generating the repair solution according to the repair strategy and synthesizing repaired configurations.

In our production environments, applications are classified into different priorities, and our engineers prefer to guarantee the service quality of critical applications as fast as possible, that is, the deployment of critical applications should be prioritized, and the repair should not introduce more potential conflicts. To meet the goal, we expect the repair strategy to be simple, intuitive, easy to understand, and scalable for various, potential changing, deployment failures in the complicated environments of a large data center. Thus, the default repair strategy of DeployFix is to remove the constraints of the applications with lower priorities in the MUC set. When multiple applications have the same priority, DeployFix randomly selects the constraints to remove. With constraints

c0 appA
c1 ∧ (¬ (app=appA ∧ performance=high-performance))
c2 ∧ (app=appA → app=appB)
c3 ∧ (app=appB → performance=high-performance)

UNSAT

(a) Formula of appA on a high-performance node

appa = True

C_1: ¬ (True ∧ p=high) = True.

⇒ p=high = False.

C_2 : True → app=appB = True

UNSAT

⇒ app=appB = True

C_3: True → p=high = True

p=high = True

MUC: 分析不满足用的最少条件.

c0 appB
c1 ∧ (¬ (app=appA ∧ performance=high-performance))
c2 ∧ (app=appA → app=appB)
c3 ∧ (app=appB → performance=high-performance)

SAT

(b) Formula of appB on a high-performance node

可行解

{app=appB: True, app=appA: False
p=high: True }.

c0: app B= True

C_1 ¬ (app=APPA ∧ P=high) =True.

app=appA ∧ p=high =False.

① app=appA =false p=high = True

C_2: False → True = True

C_3: True → True = True

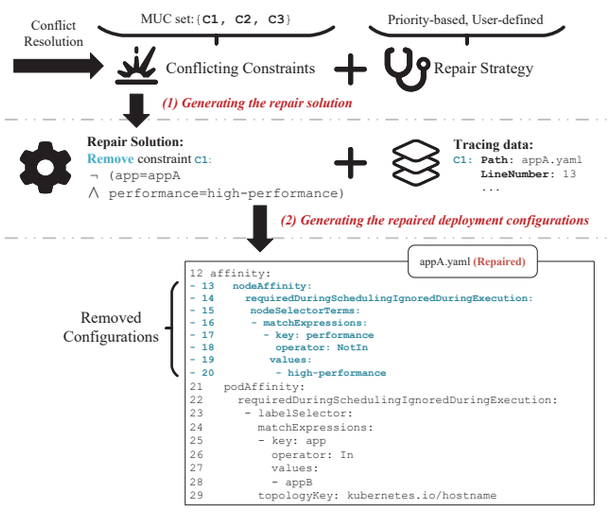


Figure 6: The code generation process for repairing the conflicts in the motivating example of Figure 1. The generated deployment configuration file removes constraint C1 - lines 13-20, according to the priority-based repair strategy.

to remove selected, DeployFix synthesizes repaired deployment configuration files as the automatic solution that can be directly applied to the scheduler according to the tracing data in DeployIR. Figure 6 demonstrates how DeployFix removes conflicting constraint C1 from the deployment configurations of appA in Figure 1 and synthesizes the repaired deployment configuration.

4 Evaluation

We evaluate DeployFix in both simulation and production environments to answer the following research questions:

RQ1. Does DeployFix correctly identify conflicting constraints and generate repair solutions?

RQ2. How fast is DeployFix in repairing deployment failures caused by conflicting deployment constraints?

RQ3. How scalable is DeployFix in repairing diverse deployment failures in practice?

4.1 Evaluation in Simulation Environments

4.1.1 Experimental Setup We build two types of test cases in the simulation environments: (a) *human-synthesized test cases*, which are used to verify the correctness of the satisfiability judgment of DeployFix and the ability to identify the conflicting constraints in the deployment configurations, and (b) *randomly-generated test cases*, which are used to construct a large-scale test set for further evaluation of the efficiency.

The human-synthesized cases include the cases of a single application with self-affinity, self-anti-affinity, and both self-affinity and self-anti-affinity constraints and the cases of applications with affinity or anti-affinity constraints among applications and nodes, which cover the most basic cases of deployment constraints and the combination of these constraints. The number of applications and nodes in these test cases is not more than three, with one to ten constraints in each case. The dataset has 40 the human-synthesized test cases in total. These 40 cases do not fully cover all the cases when the number of applications is less than or equal to three, further emphasizing the difficulty of identifying conflicting constraints as the number of applications increases.

Table 2: Simulation Experiment Results

Condition	Baseline	DeployFix
Human-Synthesized	≤ 3 minute	≤ 1 second
Randomly-Generated	≥ 1 hour[1]	≤ 1 minute

[1] The simulation cannot provide a repair solution, so the time is when Kowk starts scheduling to the time when the scheduling results no longer change, without troubleshooting or repair.

In the randomly-generated cases, several Python scripts are used to generate deployment configurations of applications with multiple deployment constraints. The configurations involve more than 100 applications and nodes, and each application has not more than ten deployment constraints, with about 100 to 300 constraints in each case. In the experiments, we generate 10 groups of randomly-generated test cases. The generated simulation cases cover most of the use cases we encountered in practice.

4.1.2 Baselines We evaluate DeployFix with two baseline methods. (a) *manual verification*: Our engineers manually inspect the satisfiability of constraints and identify the conflicting constraints in the configurations of human-synthesized test cases. The baseline time is the average time from distributing each test case to several engineers to the final repair solution. (b) *simulation cluster*: We use KWOK [53] to simulate a Kubernetes cluster and import the randomly-generated deployment configurations.[1] When there are conflicts, a subset of the conflicting applications will always be Pending. We check whether the applications that are Pending in the simulated cluster match the results of DeployFix. For the simulation experiments, we repeat the experiments 10 times. The baseline is the time Kwok takes from the start of scheduling to the time when the scheduling results no longer change. It is worth noting that the simulation cannot provide a repair solution. The time of DeployFix is the average time from inputting the configurations to obtaining the repaired deployment configurations for different test cases.

4.1.3 Results To answer **RQ1**, in both conditions, the results of DeployFix are consistent with (1) the manually identified root causes, (2) the applications that are Pending in the simulated cluster. Also, the repair solution of DeployFix makes the unschedulable applications schedulable again. According to the comparison of the time consumption in Table 2, in both conditions, DeployFix shows a significant efficiency improvement over manual verification or simulation, which answers **RQ2**.

4.2 Evaluation in Production Environments

4.2.1 Background It is practically impossible for a simulated cluster to automatically identify the conflicting deployment constraints

[1] Due to the rate limit of Kwok, it could take a lot of time to wait for all scheduling decisions finished. According to our experiments, we find that it could take hours to complete all scheduling decisions in large-scale test cases. To simplify the test in the artifact, we set the waiting time to 60s by default, and the scheduler could not have finished yet. When checking, we focus on whether the results given by DeployFix are a subset of the applications that are still Pending. In summary, all pods of conflicting applications are Pending but not all Pending pods are conflicting.

Cases	Impact	Baseline[1]	DeployFix[2]
Case 1. An invalid affinity constraint was added to app1 due to a spelling mistake, which is used to assign the instances of the application to a specific node.	The constraint has no use and blocks the deployment of app1 since there is no such node available. The rolling update or scaling up can not be performed as expected.	≤ 15 minutes	≤ 1 minute
Case 2. An invalid anti-affinity constraint was added to app1 due to a spelling mistake, which is used to avoid assigning the instances of the application to a specific node.	The performance of app1 downgrades due to resource competition or being scheduled to inappropriate nodes. However, the deployment of the application still performs "normally". It could take a long time to discover after the misconfiguration.	≤ 15 minutes	≤ 1 minute
Case 3. A constraint that conflicts with the existing constraints was added to app1. The constraint is enabled only during scheduling.	The new instance of app1 can be blocked from being scheduled. The rolling update or scaling up can not be performed as expected.	≤ 15 minutes	≤ 1 minute
Case 4. A constraint that conflicts with the existing constraints was added to app1. The constraint is enabled during both scheduling and execution.	The running instances of app1 could be evicted out. In extreme cases, there are no instances of app1 available, causing no available instances to handle the incoming requests.	≤ 15 minutes	≤ 1 minute
Case 5. An anti-affinity constraint that conflicts with the existing constraints is added to app1. The constraint is enabled during scheduling. The instances of app1 can be deployed normally, but the applications that depend on app1 can not be scheduled as expected.	Some applications that depend on app1 can not be scheduled as expected when they need to be scaled up after several hours to several days. Due to the cascading effect, it could take more time to identify the conflicting constraints, since there is no change in these configurations of the applications themselves.	several hours	≤ 1 minute
Case 6. Many deployment constraints conflicting with existing constraints are added to the scheduler.	It could take a lot of time and effort for engineers to manually troubleshoot and repair possible conflicts.	several days	≤ 1 minute

Table 3 详细列出了这 6 种从真实故障中抽象出的常见案例，这部分是本节的核心。

Case	案例描述	影响	Baseline (人工)	DeployFix
Case 1	**亲和性约束拼写错误**: `app1` 因一个无效的亲和性约束（如目标节点标签拼错）无法调度到特定节点。	应用无法滚动更新或扩容。	**≤ 15 分钟**	**≤ 1 分钟**
Case 2	**反亲和性约束拼写错误**: `app1` 因一个无效的反亲和性约束被调度到不合适的节点。	应用性能可能因资源竞争而下降，但部署本身是“正常”的，问题难以被立即发现。	**≤ 15 分钟**	**≤ 1 分钟**
Case 3	**仅调度时生效的冲突**: 为 `app1` 添加了一个与现有约束冲突的新约束，该约束仅在调度时生效。	新的 `app1` 实例无法调度，导致滚动更新或扩容失败。	**≤ 15 分钟**	**≤ 1 分钟**
Case 4	**调度和执行时都生效的冲突**: 为 `app1` 添加了在调度和执行期间都生效的冲突约束。	运行中的 `app1` 实例可能被驱逐，极端情况下可能导致服务完全不可用。	**≤ 15 分钟**	**≤ 1 分钟**
Case 5	**级联效应导致的冲突**: 为 `app1` 添加了一个反亲和性约束。 `app1` 自身部署正常，但依赖于 `app1` 的**其他应用**在数小时或数天后扩容时失败。	由于问题根源不在直接受影响的应用配置中，**排查极其耗时**。	**数小时**	**≤ 1 分钟**
Case 6	**大规模/批量冲突**: 在集群迁移或调度器升级时，大量不兼容的规则被一次性添加。	工程师需要花费大量时间和精力来手动排查和修复可能存在的冲突。	**数天**	**≤ 1 分钟**

or provide a repair solution. The randomly-generated test cases may not reflect real conditions in the production environments. To better demonstrate the correctness and scalability of DeployFix, we further evaluate DeployFix in a production data center at Alibaba, one of the largest e-commerce providers in China. There are tens of thousands of nodes and applications and hundreds of thousands of instances of applications, involving e-commerce, databases, big data processing, and large models.

4.2.2 Baselines We replayed the past deployment failures caused by the conflicting deployment constraints. We abstracted 6 common cases that DeployFix can solve, accounting for 80% of the deployment failures in our production environments. The experimental results show that DeployFix's repair solutions match the finally-deployed solutions in the incident reports or make the unschedulable application schedulable.

To compare the efficiency and scalability, it is difficult to give a unified time comparison due to the various numbers of applications and the scale of constraints involved in different failures. Thus, the baseline is the approximate time from the engineers taking over the failures to the completion of the repairs recorded in the reports. Completion usually means that the applications with higher priorities can be scheduled, while the applications with lower priorities could take a longer time to be schedulable again. The time of DeployFix is from inputting the configurations to obtaining the repaired deployment configurations. Each time is the average of multiple incident reports.

4.2.3 Case Studies Table 3 lists the key concerns of real incident reports and the frequently-happened cases. We believe that they help readers understand the production environments.[1] As shown in it, when an affinity constraint is incorrectly applied to the scheduler due to spelling mistakes (referred to as **Case 1**), it could cause the application to be unscheduled due to the inability to find the corresponding nodes. In this case, the conflicting condition can also be quickly located through manual inspection based on the change history. However, if the constraint is anti-affinity, it could only cause the performance degradation of the application (referred to as **Case 2**), which could take a long time to notice the degradation. DeployFix tracks the definition of every entity and reports conflicts for the entities that are not found in the configurations, for reasons like spelling mistakes or missing nodes with corresponding labels.

When a new constraint that enables only during scheduling is added and the rolling update is launched immediately, it will immediately cause troubles and the conflicting condition can be also quickly located based on the change history (referred to as **Case 3**). Similarly, if the constraints are enabled during execution, the running instances will be evicted. In extreme cases, there are no instances to serve any incoming requests (referred to as **Case 4**). However, if the change is not related to the application itself but to the dynamic environments, it could be time-consuming to identify the problem due to the cascading effects (referred to as **Case 5**).

When a large number of rules are added to the deployment configurations at once. For example, when applications are migrated across clusters or the scheduler is upgraded, it could take quite a lot of time and effort for cloud engineers to identify and repair these conflicts (referred to as **Case 6**). According to the failure reports in practice, these failures usually take several days for our engineers to manually inspect and repair.

4.2.4 Results To answer **RQ1**, according to the comparison with the finally-deployed solutions recorded in the incident reports, the

[1]Due to the confidential restrictions, the details can not be fully disclosed.

solution provided by DeployFix matches the solutions in all six cases, showing DeployFix can correctly repair the deployment failures caused by conflicting constraints. To answer **RQ2**, compared with manual inspection for all the cases, DeployFix can troubleshoot and repair these conflicts within minutes, which has greatly improved the efficiency. Moreover, configurations can be verified before being applied to the scheduler, thus early preventing and discovering conflicting constraints. To answer **RQ3**, **Case 1** ~ **Case 6** involve 80% of the deployment failures caused by conflicting deployment constraints we have encountered in practice, proving the scalability of DeployFix for different cases. **Case 1** and **Case 2** are the most frequent types of errors, which account for 40% of our failure reports. **Case 3** ~ **Case 5** account for a smaller proportion, but when they occur, they take longer time to manually inspect and repair than the first two cases. **Case 6** is relatively less common. The remaining 20% of the cases involve some customized implementations of the internal scheduler, which are not part of the official implementation of Kubernetes. The above results show that DeployFix can effectively simplify the complexity of troubleshooting and repairing conflicting constraints in practice, relieving the burden of cloud engineers, helping avoid deployment failures and reducing reputational and financial losses.

4.3 Threats to Validity

To enhance external validity, we use three variants of the configuration sets, including manually constructed, randomly-generated test cases, and real datasets from our production environments. All cases show that DeployFix can automatically repair deployment failures with efficiency and scalability. However, we are also aware that the experimental results could not cover all the possible use cases and could not automatically be transferable to all other cases. We believe that our production environments have enough representativeness to cover most cases in practice.

To enhance internal validity, we formalize the satisfiability problem of deployment constraints in propositional logic. Furthermore, we use the widely used Z3 Solver [47] in both academia and industry to check the satisfiability while guaranteeing the correctness and efficiency of resolving conflicting constraints. Moreover, our implementations and experiments include the two most widely used schedulers - Kubernetes and Apache Yarn.

4.4 Discussion

DeployFix currently supports both the dynamic repair of deployment failures and input buggy deployment configurations, providing the repaired configuration files. Given the diversity and complexity of existing schedulers, the prototype version of DeployFix supports most syntaxes of Kubernetes except `taints & tolerations` and `nodeSelector`[2] and major syntaxes of Apache Yarn except `Cardinity`. The supports are based on our practical requirements. Meanwhile, the formalization of constraints in propositional logic and the use of SAT solvers still have certain limitations. However, in the production environments we have encountered so far, DeployFix and SAT are sufficient to cover most cases. On the other hand, engineers will not "blindly" accept the solution provided by DeployFix, but inspect it anyway. They will check whether the solution provided by DeployFix meets their expectations and accept the solution if there is no issue after the inspection, indicating possible limitations of the default repair strategy. In this section, we discuss these two issues in more detail.

4.4.1 Semantic Enhancement The formalization of deployment constraints in propositional logic and solving with SAT solvers lack the semantics of the number of instances of applications and nodes, making the number of variables and constraints does not increase with the number of instances and nodes. This makes it possible to provide solutions in a short time to better support our practical requirements. However, it also makes DeployFix have drawbacks in handling some cases. For example, the internal implementation of our scheduler can specify that a self-anti-affinity constraint can be violated *n* times, indicating that at most *n* instances can be deployed to the node, but DeployFix cannot express it, which is one of the 20% of the cases that DeployFix cannot handle. Since this is not part of the official implementation of Kubernetes, DeployFix does not consider it for now. We believe that using a first-order logic and satisfiability modulo theory (SMT) solver or using a linear equation system and integer linear programming (ILP) solver could enhance semantic expressiveness.

4.4.2 Solution Selection The default strategy of DeployFix is to prioritize the deployment of the critical applications. We hope that it is clear and helpful to understand the design of DeployFix in the presentation. Also, the strategy performs well in our production environments according to our practical experience. However, removing some constraints could cause the application unschedulable, e.g., an application requires a node with GPUs. Marking these constraints as "critical" bans DeployFix from removing them.

Moreover, we are aware that the practical requirements could be diverse and the default strategy may not perform as well as we expected in all conditions. For example, the default strategy defines the critical applications based on the priority of the applications, while some users may treat the applications with more running instances as the critical applications. In addition, users may want to change the fields in the constraints or use a large model to generate fixed configurations, instead of directly removing the conflicting constraints. To support the diverse requirements, DeployFix allows users to change the repair strategy and expects to integrate more users' requirements as plugins to help users select different solutions more fluently.

5 Related Work

APR aims to automatically fix defects in programs without human intervention and has long been a hot topic in software engineering. The workflow of APR usually involves *defect localization*, *patch generation*, and *patch validation*. GenProg [10] is the first APR tool that uses genetic programming to evolve a program to repair defects. PAR [17] uses the fix templates mined from history patches to generate patches. AllRepair [18] converts the program into Static Single Assignment (SSA) form by unrolling the loop and recursion and uses incremental SAT and SMT solvers to restrict the search space and generate feasible patches. SemFix [15] and DirectFix [16] uses symbolic execution and SMT solvers to collect the semantic

[2]The official recommendation of Kubernetes is to use `nodeAffinity` instead of `nodeSelector`.

constraints of the defects and generate patches using component-based program synthesis. S3 [19] combines syntax- and semantic-guided ranking features by abstract syntax tree (AST) difference and cosine similarity to effectively synthesize high-quality repairs. ARJA [11] extends GenProg and transforms the patch generation problem into a multi-objective optimization problem to minimize the number of patches and maximize the number of passed test cases after patching. Some other studies focus on the automatic repair of configuration bugs. ConfError [29] injects misspelling, structure, and semantic errors into the configuration files to test the robustness of the system. ECCFixer [32] leverages Reiter's theory [30] to map the configuration value problem to an SAT and solve the minimal set of variables that need to be reset and modified from the constraints and configuration sets. Spex [26] mines the type and value constraints, control dependencies, and value correlations through static analysis and generates tests violating these constraints to evaluate the robustness when the misconfiguration happens. Parachute [54] conducted an in-depth study on the configuration bugs due to hot updates and mutated existing configurations to identify improper handling of configuration changes by differential analysis. DiagConfig [34] focuses on conducting configuration diagnosis of performance violations by static analysis to track option propagation and identify performance-sensitive options. Most APR and configuration bug repair studies are *static repair*, without the support for dynamic deployment environments, making them unable to troubleshoot and repair deployment failures. Compared to them, DeployFix supports the dynamic repair of deployment failure by including running nodes when translation and expressing them uniformly when formalization.

Cloud scheduler helps maximize resource utilization and minimize resource fragmentation while meeting the requirements of both the cloud providers and the users. A common requirement is to assign certain applications to or not to the same nodes, called deployment constraints, aiming at reducing network transmission latency, improving data locality, avoiding resource contentions, or enhancing fault tolerance. Google Borg [55] or Kubernetes [55] supports different kinds of constraints from different perspectives, including affinity and anti-affinity. Through these constraints, Borg and Kubernetes can express a wide range of placement requirements to offer an approximate solution for maximizing resource utilization and minimizing resource fragmentation. TetriSched [56] offers various placement and time constraints but requires exact machines to be specified. TetriSched also uses ILP to solve the orchestration problem, but it is not scalable to large-scale data centers since it handles all scheduling decisions at once. Firmament [57] leverages a graph-based centralized scheduler and supports application affinity to specific machines. However, adding more constraints to Firmament introduces extra vertices and edges to the graph, which increases the complexity of the graph and the scheduling latencies. MEDEA [58] introduced powerful deployment constraints with formal semantics to express the placement requirements of LRAs, aiming at achieving goals like maximizing the number of deployed LRAs and minimizing the number of violations or resource fragmentation. George [59] proposed a reinforcement learning-based approach to place LRAs in a data center. It designs a novel projection-based proximal strategy optimization algorithm in combination with an integer linear optimization technique to intelligently schedule LRA containers under constraints. Alibaba's Optum [60] is a unified scheduling system that supports transforming scheduling requests from different schedulers into a unified format and scheduling them uniformly.

These schedulers all support complex deployment constraints in various syntaxes but lack enough tactics to maintain these constraints. As aforementioned, there is currently no automated tool to repair the deployment failures caused by conflicting deployment constraints. DeployFix fills the gap in this field. With DeployIR, DeployFix supports translating constraints written in different syntaxes into a unified representation and reuses the subsequent steps, simplifying the complexity of extending the support for more syntaxes and schedulers.

6 Conclusion

We proposed DeployFix, an approach that supports not only static repair but also dynamic repair of deployment failure in a large data center. We defined the DeployIR to enable the translation of deployment configurations written in heterogeneous syntaxes. To address the complex root cause analysis, DeployFix reduces the diagnosis of deployment failures to conflict resolution and uses the off-the-shelf solver to identify the MUC set, the root cause of the deployment failure. With the MUC set and the tracing data, DeployFix generates repaired configurations as the automatic solution.

We implemented DeployFix in Rust and made the source code publicly available. DeployFix supports two widely-used schedulers and their syntaxes: Kubernetes and Apache Yarn. DeployFix uses the off-the-shelf Z3 solver to identify the conflicting constraints, which helps to guarantee the correctness and efficiency of the root cause analysis.

We demonstrated the correctness, efficiency and scalability of DeployFix in both simulation and production environments. The experimental results show that DeployFix outperforms the state-of-the-art and can correctly repair the deployment failures. DeployFix is efficient and scalable to be used in large-scale cloud data centers. The automatic repair solution of DeployFix greatly relieves the burden of cloud engineers to diagnose and repair deployment failures and enhances the reliability and stability of the cloud services. We believe that DeployFix solves a changing problem that has not been well addressed by previous work and makes real contributions to the service reliability and stability in large data centers.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (62272167), Alibaba Group, and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

References

[1] Google Cloud. Google Cloud: Google Cloud Service Health. https://status.cloud.google.com/summary.
[2] Amazon Web Services. Amazon Web Services.: Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/12721/.
[3] Alibaba Cloud. Alibaba Cloud: Notices of the Alibaba Cloud Service, 2023.
[4] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343. ACM, 2015.

[5] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, page 1, USA, 2003. USENIX Association.

[6] Yifan Wu, Bingxu Chai, Ying Li, Bingchang Liu, Jianguo Li, Yong Yang, and Wei Jiang. An empirical study on change-induced incidents of online service systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 234–245, 2023.

[7] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 126–141, New York, NY, USA, 2022. ACM.

[8] Xiaoyun Li, Guangba Yu, Pengfei Chen, Hongyang Chen, and Zhekang Chen. Going through the life cycle of faults in clouds: Guidelines on fault handling. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering*, pages 121–132, 2022.

[9] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. Security misconfigurations in open source kubernetes manifests: An empirical study. 32(4), may 2023.

[10] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[11] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067, 2020.

[12] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 535–547, 2023.

[13] Welder Pinheiro Luz, Gustavo Pinto, and Rodrigo Bonifácio. Adopting DevOps in the real world: A theory, a model, and a case study. *Journal of Systems and Software*, 157:110384, 2019.

[14] Tao Ji, Liqian Chen, Xiaoguang Mao, and Xin Yi. Automated program repair by using similar code containing fix ingredients. *2016 IEEE 40th Annual Computer Software and Applications Conference*, 1:197–202, 2016.

[15] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781. IEEE Computer Society, 2013.

[16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 448–458. IEEE, 2015.

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 802–811. IEEE Press, 2013.

[18] Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 593–611, Cham, 2016. Springer International Publishing.

[19] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604, Paderborn Germany, August 2017. ACM.

[20] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 344–358. ACM, 2021.

[21] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 955–970, New York, NY, USA, 2020. ACM.

[22] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: Reduced instruction footprint for Coverage-Guided fuzzing. In *2021 USENIX Annual Technical Conference*, pages 147–159. USENIX Association, July 2021.

[23] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 858–870, New York, NY, USA, 2021. ACM.

[24] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: efficient fuzz testing for data analytics using framework abstraction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 722–733, New York, NY, USA, 2021. ACM.

[25] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. Challenges and opportunities: an in-depth empirical study on configuration error injection testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 478–490, New York, NY, USA, 2021. ACM.

[26] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 244–259, New York, NY, USA, 2013. ACM.

[27] Sai Zhang and Michael D. Ernst. Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 12–23, New York, NY, USA, 2015. ACM.

[28] Shanshan Li, Wang Li, Xiangke Liao, Shaoliang Peng, Shulin Zhou, Zhouyang Jia, and Teng Wang. ConfVD: System Reactions Analysis and Evaluation Through Misconfiguration Injection. *IEEE Transactions on Reliability*, 67(4):1393–1405, 2018.

[29] Lorenzo Keller, Prasang Upadhyaya, and George Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*, pages 157–166. IEEE, 2008.

[30] Sai Zhang. Confdiagnoser: An automated configuration error diagnosis tool for java software. *2013 35th International Conference on Software Engineering*, pages 1438–1440, 2013.

[31] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. Ecfuzz: Effective configuration fuzzing for large-scale systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. ACM.

[32] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *2012 34th International Conference on Software Engineering*, pages 58–68, 2012.

[33] Chengpeng Wang, Gang Fan, Peisen Yao, Fuxiong Pan, and Charles Zhang. Verifying data constraint equivalence in FinTech systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering*, pages 1329–1341. IEEE, 2023.

[34] Zhiming Chen, Pengfei Chen, Peipei Wang, Guangba Yu, Zilong He, and Genting Mai. Diagconfig: Configuration diagnosis of performance violations in configurable software systems. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 566–578, New York, NY, USA, 2023. ACM.

[35] Wolfgang Mayer, Rajesh Thiagarajan, and Markus Stumptner. Service Composition as Generative Constraint Satisfaction. In *2009 IEEE International Conference on Web Services*, pages 888–895. IEEE, 2009.

[36] Dong Yang and Ming Dong. A constraint satisfaction approach to resolving product configuration conflicts. *Advanced Engineering Informatics*, 26:592–602, 08 2012.

[37] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 517–528. IEEE Press, 2015.

[38] Manage Kubernetes configuration at scale using the new GitOps observability dashboard, 2024.

[39] Configuring Pods to use a Kubernetes service account - Amazon EKS, 2024.

[40] MGoedtel. Resource management best practices for Azure Kubernetes Service (AKS) - Azure Kubernetes Service, May 2023.

[41] Configuration Best Practices, 2024.

[42] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, and Liping Zhang. Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11. ACM, 2022.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17. ACM, 2015.

[44] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181. ACM, 2021.

[45] Shuang Chen, Christina Delimitrou, and José F. Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, Providence RI USA, April 2019. ACM.

[46] Yuhang Liu, Xin Deng, Jiapeng Zhou, Mingyu Chen, and Yungang Bao. Ah-Q: Quantifying and Handling the Interference within a Datacenter from a System Perspective. In *2023 IEEE International Symposium on High-Performance Computer Architecture*, pages 471–484. IEEE, 2023.

[47] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 337–340, 04 2008.

[48] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 544–558. Springer Berlin Heidelberg, 2007.
[49] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.
[50] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, sep 2011.
[51] E. Tsang. *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, 1993.
[52] Donald B. Johnson. Finding all the elementary circuits of a directed graph. 4(1):77–84, mar 1975.
[53] Kubernetes SIGs. Kwok (kubernetes without kubelet). https://github.com/kubernetes-sigs/kwok.
[54] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. Understanding and Detecting On-The-Fly Configuration Bugs. In *2023 IEEE/ACM 45th International Conference on Software Engineering*, pages 628–639. IEEE, 2023.
[55] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17. ACM, 2015.
[56] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16. ACM, 2016.
[57] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: fast, centralized cluster scheduling at scale. *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, page 99–115, 2016.
[58] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. M EDEA: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13. ACM, 2018.
[59] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 258–272, Seattle WA USA, November 2021. ACM.
[60] Chengzhi Lu, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. Understanding and optimizing workloads for unified resource management in large cloud platforms. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 416–432, New York, NY, USA, 2023. ACM.