

UNIVERSITY OF CALIFORNIA SAN DIEGO

Improving the Dependability of Python-Based Database-Backed Web Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Haochen Huang

Committee in charge:

Professor Yuanyuan Zhou, Chair
Professor Arun Kumar
Professor Deian Stefan
Professor Geoffrey M. Voelker
Professor Xinyu Zhang

2022

Copyright
Haochen Huang, 2022
All rights reserved.

The dissertation of Haochen Huang is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

Dedicated to my family.

TABLE OF CONTENTS

Dissertation Approval Page	iii
Dedication	iv
Table of Contents	v
List of Figures	vii
List of Tables	viii
Acknowledgements	ix
Vita	xi
Abstract of the Dissertation	xii
Chapter 1	
Introduction	1
1.1 Motivation	1
1.2 Contribution	3
1.2.1 CFINDER: Protecting Data Integrity With Inferred Database Constraints	4
1.2.2 PYLIVE: On-The-Fly Code Change	5
Chapter 2	
CFINDER: Protecting Data Integrity With Inferred Database Constraints	7
2.1 Introduction	7
2.1.1 Problem: Missing Database Constraints	7
2.1.2 Consequences of Missing Constraints	9
2.1.3 Why DB Constraints Are Better Guards?	10
2.1.4 Our Contributions	12
2.2 Understanding Missing Database Constraints in Web Applications	13
2.3 Design and implementation	16
2.3.1 Design Choices: Possible Ways to Find Missing Constraints	16
2.3.2 CFINDER Overview	18
2.3.3 Code Patterns with Assumptions on DB Constraints	19
2.3.4 Code Patterns Detection Algorithm	24
2.3.5 Database Constraints Extraction	26
2.4 Evaluation	28
2.4.1 Effectiveness in Detecting <i>Missing</i> DB Constraints	30
2.4.2 False Positives in Detected Missing DB Constraints	32
2.4.3 Coverage of Database Constraints	33
2.4.4 Performance of CFINDER	36
2.4.5 Developers' Feedback Discussion	36

	2.5	Related Work	37
	2.6	Limitation & Discussion	39
	2.7	Acknowledgments	40
Chapter 3		PYLIVE: On-The-Fly Code Change	41
	3.1	Introduction	41
	3.2	Background	44
	3.3	PYLIVE Framework	45
	3.3.1	Design Objectives	46
	3.3.2	PYLIVE's Interfaces	47
	3.3.3	Support Dynamic Changes	49
	3.3.4	Identify Safe Change Point	50
	3.3.5	Support for Multi-threads and Multi-processes	53
	3.4	Use Cases	54
	3.4.1	On-the-fly Logging for diagnosis	54
	3.4.2	On-the-fly Profiling	55
	3.4.3	Dynamic Patching	56
	3.5	Evaluation	57
	3.5.1	Methodology	57
	3.5.2	Overall Performance Results	59
	3.5.3	Case Studies	60
	3.5.4	Human Effort	64
	3.6	Limitations and Discussion	65
	3.7	Related Work	67
	3.8	Acknowledgments	71
Chapter 4		Conclusion	72
	4.1	Lessons Learned	73
Bibliography		75

LIST OF FIGURES

Figure 2.1:	Three real-world issues [136, 168, 63] that violated three types of DB constraints and led to severe consequences.	8
Figure 2.2:	Comparison between applications with and without database constraints. . .	10
Figure 2.3:	A real-world issue [62] from Oscar caused by missing code validations. . .	11
Figure 2.4:	Code snippets with implicit assumptions on database constraints.	13
Figure 2.5:	The overview of CFINDER. CFINDER contains three steps to infer the missing database constraints from the application source code. The green boxes are the output of the steps.	18
Figure 2.6:	Code patterns with implicit assumptions on three DB constraint types, together with real-world examples and explanations.	20
Figure 2.6:	Code patterns with implicit assumptions on three DB constraint types, together with real-world examples and explanations.(cont.)	21
Figure 2.7:	Example of pre-defined syntax tree patterns. We use them to match with the candidate syntax trees.	24
Figure 2.8:	Matching syntax tree T (left) with pre-defined syntax pattern P (right). . .	26
Figure 2.9:	Infer the constraint table from code with two steps	28
Figure 3.1:	An example of unsafe change points for a patch from Django [11].	52
Figure 3.2:	An example of state check function for the patch in figure 3.1	52
Figure 3.3:	PYLIVE’s dynamic logging spec for an urgent, real world bug in Shuup e-commerce system [28].	55
Figure 3.4:	On-the-fly profiling using PYLIVE to diagnose a critical performance issue occurred in Oscar e-commerce system [21].	56
Figure 3.5:	A real world security patch to Django [1] and PYLIVE’s dynamic change spec for it.	57
Figure 3.6:	Throughput comparison of three on-the-fly logging cases and three on-the-fly profiling cases with PYLIVE in comparison with today’s practices—stop and restart with logging added and profiling enabled.	61
Figure 3.6:	Throughput comparison of three on-the-fly logging cases and three on-the-fly profiling cases with PYLIVE in comparison with today’s practices—stop and restart with logging added and profiling enabled.(cont.)	62
Figure 3.7:	Throughput comparison of two representative patching cases with PYLIVE and restarting services.	65

LIST OF TABLES

Table 1.1:	Popular Python-based frameworks and applications for online services. . . .	2
Table 2.1:	The web applications used in our study.	14
Table 2.2:	The number of database constraints that are missed first and added in later pull requests in each application.	15
Table 2.3:	Reasons why developers add the <i>missing</i> constraints.	16
Table 2.4:	Evaluated applications and detected missing DB constraints from them. . . .	30
Table 2.5:	Examples of confirmed missing database constraints.	31
Table 2.6:	The breakdowns of the number of detected <i>missing</i> database constraints for each constraint type and code pattern.	32
Table 2.7:	The precision of detected missing constraints by CFINDER.	34
Table 2.8:	The percentage of <i>existing</i> constraints already set in the database that CFINDER can cover in the detection.	35
Table 2.9:	The percentage of <i>missing</i> constraints that CFINDER can cover in the collected dataset.	36
Table 2.10:	Time (seconds) to run CFINDER’s static analysis.	36
Table 3.1:	20 real-world cases evaluated in our experiments.	58
Table 3.2:	Lines of code (LOC) of change specification for PYLIVE.	64

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Professor Yuanyuan (YY) Zhou for her tremendous support and advice throughout my Ph.D. journey. She guided me on how to be a “system” researcher. I learned how to identify real-world problems worth solving and gain unique insights; I learned the standard of top-level system research: a good research problem, intellectual-challenging methodologies, solid evaluations, good writing, and presentation, etc. Every week’s group meeting with her has been a challenging and fruitful adventure. Not only research, but I also learned so many valuable life lessons from her that have dramatically changed the way I see and behave in the world. She taught me to face the real world and be tough when facing failures; She taught me to be my best, to seize every opportunity, and to fight for what I want; She taught me to pursue what I truly believe in and am passionate for; and I believe I will continue to benefit from these lessons for the rest of my life.

I then would like to thank my thesis committee members, Professor Arun Kumar, Professor Geoffrey Voelker, Professor Deian Stefan, and Professor Xinyu Zhang. Their insightful feedback and suggestions greatly improve this dissertation.

I would like to thank other professors and staff in the SysNet group. Professor Geoffrey Voelker is always being so helpful and inspiring. He provided insightful advice for many of my presentations and research projects. I would also like to thank Professor Stefan Savage, George Porter, Yiyang Zhang, and Alex Snoeren for holding system courses, syslunch and 294 seminars, and gave me valuable feedback.

I would like to thank Tianwei Sheng, my supervisor during my internships at Whova company for giving me the great opportunity. He guided me through exciting projects and gave me the freedom to explore more. The invaluable experience gave me the insight into the real backend and infrastructure in the industry and gave me a solid understanding of the problems in system research. I would also like to thank my coworkers at Whova: Andrew Yoo, Xinxin Jin, Sihang Li, etc. It was a pleasure working with them and I learned a lot from them.

I would like to thank my lab mates in the Opera group. Chengcheng Xiang gave me tremendous help in both research and life when I was in my junior year, and he set an example for me. We collaborated on multiple projects and he taught me a lot hand-over-hand. Bingyu Shen was another role model and friend in my life. He helped me with the experiment evaluations in the second project. He always provided insightful discussions and lessons. I have also worked closely with Li Zhong on several projects, and I enjoyed the discussions with her. Their hard work contributed a lot to the projects presented in this dissertation and some other projects I've worked on, which made the completion of this thesis possible. I also want to thank my other lab mates and many others in the SysNet group for their valuable discussions and feedback: Yudong Wu, Tianyi Shan, Eric Mugnier, Mingyao Shen, Vector Li, etc.

Finally, I want to thank my parents, Qianfan Huang and Jiexia Lu, who always give me their full support and unconditional love. I dedicate this dissertation to my family. I would also thank my roommates Zijin Lin, Jiayuan Gu, and Yutong Shao, we went through colorful Ph.D. journeys together these years. Their company means a lot to me, especially during the pandemic.

Chapter 2, in full, is a reprint of the material as it appears in the 28th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23) Haochen Huang, Bingyu Shen, Li Zhong, Yuanyuan Zhou, "Protecting Data Integrity of Web Applications With Database Constraints Inferred From Application Code". The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in the 2021 USENIX Annual Technical Conference (ATC'21) Haochen Huang, Chengcheng Xiang (co-first authors), Li Zhong, Yuanyuan Zhou, "PYLIVE: On-the-Fly Code Change for Python-based Online Services". The dissertation author was the primary investigator and author of this paper.

VITA

- 2018 B. S. in Computer Science, Shanghai Jiao Tong University
- 2022 M. S. in Computer Science, University of California San Diego
- 2022 Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Haochen Huang, Bingyu Shen, Li Zhong, Yuanyuan Zhou, “Protecting Data Integrity of Web Applications With Database Constraints Inferred From Application Code”, *Proceedings of the 28th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’23)*, March, 2023..

Haochen Huang, Chengcheng Xiang (co-first authors), Li Zhong, Yuanyuan Zhou, “PYLIVE: On-the-Fly Code Change for Python-based Online Services”, *Proceedings of the 2021 USENIX Annual Technical Conference (ATC’21)*, July, 2021.

Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, Shankar Pasupathy, “PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations”, *Proceedings of the 2020 USENIX Annual Technical Conference (ATC’20)*, July, 2020..

Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, Tianwei Sheng, “Towards Continuous Access Control Validation and Forensics”, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS’19)*, November, 2019.

ABSTRACT OF THE DISSERTATION

Improving the Dependability of Python-Based Database-Backed Web Applications

by

Haochen Huang

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Yuanyuan Zhou, Chair

Python-based database-backed web frameworks have gained wide adoption in developing online web applications in many commercial companies and open-source projects. They are used in e-commerce, banking, social networking, and many aspects of our daily life. These web applications have critical requirements on high dependability: any data corruption or service downtime could lead to significant business loss. Unfortunately, such incidents caused by application bugs or operator errors remain increasingly pervasive.

This dissertation explores the potential of building and maintaining more dependable Python-based web applications by providing tooling support for developers. To achieve this, we observe the two unique characteristics of these web applications: (1) They are backed by a

database management system (DBMS), which enforces data integrity constraint checks. (2) The main application logic is developed in Python, an interpreted language with unique language features.

The first piece of this dissertation tackles the data integrity issues caused by application bugs or operator errors. Our study observes that application developers often do not take full advantage of the database-level integrity validations and miss specifying many database constraints, resulting in many severe consequences. We then build a tool called CFINDER, to automatically infer the missing database constraints from application source code by leveraging the observation that many source code patterns imply certain data constraints.

Second, the dissertation tackles the demand of applying code changes for online diagnosing and fixing while keeping the service available. We observe the unique language features of Python, i.e. meta-object protocol and dynamic typing, which makes dynamic code change much easier for Python than for C or Java. We then propose a new framework called PYLIVE, to support dynamic logging, profiling, and bug-fixing without restarting. It can be easily adopted with no modification to the runtime system.

Chapter 1

Introduction

1.1 Motivation

Python-based database-backed web applications have gained wide adoption in developing online services. Many companies build their main online services with Python. For example, Instagram’s web services and Quora’s main web services are built with Python [154, 132, 38]. Google’s Youtube front-end server is also built using Python [39, 77, 161]. In addition to commercial companies, many open-source projects also build Python-based web applications for various online services. Table 1.1 shows six categories of them, including web frameworks, e-commerce applications, content management systems, etc.

Since these online web applications are powering many important aspects of our daily lives such as e-commerce and social networking, they have high demands for dependability. We focus on two critical requirements here: high availability and data integrity. A recent survey shows that 99.99% of uptime (i.e., 52.6 minutes downtime per year) has become the minimum availability standard for most web applications [87]. A report from Statista shows that one hour of server downtime can cause more than \$301K cost for 88% of companies and more than \$5 million cost for 17% of companies [146]. Besides server downtime, data corruption [76] is another severe

Table 1.1: Popular Python-based frameworks and applications for online services.

Category	Server Frameworks (Github stars)
E-commerce	Odoo (26.8k), Saleor (17.2k), Oscar (5.5k), Shuup (1.9K)
Content management	wagtail (13.4K), django-cms (8.9K), frappe (4.2K)
Social & Team chat	Zulip (16.7K), Python Social Auth (1.7K)
Web framework	Django (67.2k), Flask (61.0k), Pyramid (3.7k)
Web Server	Gunicorn (8.6k), Tornado (20.8k), Unicorn (5.9k)
Message queue	Celery (20.4), huey (4.3k), rq (8.6k)
Cache backend	Django-cacheops (1.7k), Beaker (494)
FTP server	pyftplib (1.4k), fbtftp (367), pyrexcid (213)

threat to the online service. Any violation of data integrity can result in severe consequences, such as crashing the order placement page, corrupting the store inventory data, and blocking user login [136, 168, 63], leading to significant business loss.

As various redundancy techniques have been used in computing, storage, and network subsystems, availability and integrity issues caused by hardware errors [96, 46, 153] or crash failures [86, 53] have been reasonably well addressed in today’s data centers. Unfortunately, application bugs or operator errors cannot be solved by redundancy and are significantly understudied. They remain as increasingly pervasive root causes for a large number of incidents.

In this dissertation, we notice the developers’ demand to protect data integrity and maintain the high availability of web applications and aim to provide tooling support for them. First, to guarantee data integrity, fortunately, most of today’s relational database management systems (RDBMS) provide integrity constraints [54, 131]. Specifically, application developers specify database constraints based on their own business logic and enforce them in the database schema, such as a *not-null* constraint for `order.total`, or a *unique* constraint for `user.email`. Such database constraints would detect and refuse any incorrect data manipulation caused by either bugs in application code, or operator mistakes when directly manipulating data via the database administrator (DBA) console. Common constraints supported in popular RDBMSes include *Not-null*, *Unique*, and *Foreign key* constraints [117, 106, 127, 103]. Modern web frameworks

have supported the migration helpers for all three common database constraints in recent years, enabling applications to easily specify and enforce constraints in databases. However, despite these initiatives, many app developers still do not take full advantage of database constraints to protect their application data integrity. Therefore, it would be beneficial to help developers take full advantage of the database constraints to guide their application data integrity.

Second, when the running service shows some failures or abnormal behaviors, developers have high demands to apply the code changes to diagnose or fix it. Such code changes include adding logs to gain more diagnostic information, instrumenting programs to profile performance bottlenecks, and applying patches to fix bugs and security vulnerabilities. Applying the changes *on the fly* is necessary when diagnosing challenging issues (e.g., resource leaking or performance degradation) that are hard to reproduce in a testing environment or when patching critical bugs or security issues that need to be patched as soon as possible. More importantly, such code changes are required to cause little or no service downtime when being applied for high availability. Therefore, developers would benefit greatly from tooling support to apply code changes on-the-fly in production without restarting them.

1.2 Contribution

This dissertation seeks to provide tooling support for developers to build and maintain more dependable Python-based database-backed web frameworks. The tools are based on our observations on the *unique features* of these web applications:

- *They are backed by a database management system (DBMS), which enforces data integrity constraint checks.* When constraints are enforced by the database, even if validations are missed in some code paths, the database always acts as the final guard to perform integrity checks and detect violations against specified constraints.
- *The main application logic is developed in Python, an interpreted language with unique*

language features. Python supports the meta-object protocol [89, 22] and dynamic typing, which enables programs to dynamically modify their own metadata, including function bodies/interfaces and class attributes. This makes dynamic code change much easier for Python than for compiled languages (e.g., C/C++) and other interpreted languages that do not support the full meta-object protocol or dynamic typing (e.g., Java).

The first characteristic reminds us that web applications should take full advantage of database constraints to ensure data integrity when possible. We build a tool called CFINDER, to automatically infer the missing database constraints from application source code by leveraging the observation that many source code patterns imply certain data integrity constraints. The second characteristic motivates us to support on-the-fly logging, profiling, and bug-fixing while keeping the service available. We build a new framework called PYLIVE, which utilizes the unique language features of Python, i.e. meta-object protocol and dynamic typing. The tool can be easily adopted with no modification to the runtime system.

Thesis Statement: *By leveraging the unique features of Python-based database-backed web applications, including database support for constraint validation and Python’s language features, we can provide tooling support to help developers build and maintain more dependable web applications.*

1.2.1 CFINDER: Protecting Data Integrity With Inferred Database Constraints

Chapter 2 presents CFINDER. CFINDER is a tool that can automatically infer missing database constraints from the application source code, so that developers can take full advantage of database constraints to ensure the data integrity of their web applications.

In chapter 2, we focus on the problem of missing database constraints in web applications. We first study several widely used open-source e-commerce and communication applications,

and observe that all these applications have missed integrity constraints and many were added later as afterthoughts after issues occurred. Also, most (82%) of these cases could result in severe consequences, including page crashes and data corruption of order-related or payment-related data. Our study also shows that most (87%) issues that missed database constraints also missed code checks in some code paths in application code, indicating that solely relying on application code checks is not a safe approach to guarantee data integrity.

Motivated by the observations, we build CFINDER to automatically infer missing database constraints from the application source code by cleverly leveraging the observation that many source code patterns usually imply certain data integrity constraints. By analyzing application source code automatically, CFINDER can extract such constraints and check against their database schemas to detect missing ones.

Chapter 2 evaluates CFINDER with eight widely-deployed web applications, including one commercial company with millions of users. Overall, our tool identifies 210 previously unknown missing constraints. We have reported 92 of them to the developers of these applications, so far 75 are confirmed. Our tool achieves a precision of 78% and a recall of 79%.

1.2.2 PYLIVE: On-The-Fly Code Change

Chapter 3 presents PYLIVE. PYLIVE is a framework that aims to improve the system robustness by enabling on-the-fly code change without restarting the servers. PYLIVE can be used by developers with multiple scenarios, such as on-the-fly logging for diagnosing production-run errors, on-the-fly profiling for diagnosing production-run performance issues, and urgent dynamic patching for bugs or security patches. PYLIVE leverages the unique language features of Python, meta-object protocol, and dynamic typing, to support dynamic code change without restarting online services. PYLIVE requires no modification to the underlying runtime systems (i.e., Python interpreters), making it easy to be adopted by online services with little portability concern.

Chapter 3 evaluates PYLIVE with seven Python-based web applications that are widely

used for online services. From these applications, we collect 20 existing real-world cases, including bugs, performance issues, and patches for evaluation. PYLIVE can help resolve all the cases by providing dynamic logging, profiling and patching with little overhead. Additionally, PYLIVE also helped diagnose two new performance issues in two widely-used open-source applications.

Chapter 2

CFINDER: Protecting Data Integrity With Inferred Database Constraints

2.1 Introduction

2.1.1 Problem: Missing Database Constraints

Data integrity is critical for database-backed web applications used in e-commerce, banking, and many aspects of our daily life [76]. As various data redundancy techniques have been used in computer storage and network subsystems, integrity issues caused by hardware errors [96, 46, 153] or crash failures [86, 53] have been reasonably well addressed in today's data centers. In contrast, application bugs or operator errors are significantly understudied and remain as increasingly pervasive root causes for data integrity issues in databases [76].

Fortunately, most of today's relational database management systems (RDBMS) provide integrity constraints that help applications to guarantee desired data integrity [54, 131].

The necessity of specifying database constraints to protect data integrity has received increasing attention. For example, central players in modern web frameworks, such as Rails (Ruby), Django (Python), and Hibernate (Java), have supported the migration helpers for all

<p>Violated constraint: /*Saleor*/ One <i>Order.total</i> field has <i>null</i> value.</p> <p>Consequence: <i>[Page crash]</i> Shop admin cannot operate on the dashboard page as it crashes.</p> <p>Resolution taken: Add <i>Not-null</i> constraint to database.</p>	<p>Violated constraint: /*Zulip*/ Two <i>UserProfile.email</i> are the same, not <i>unique</i>.</p> <p>Consequence: <i>[Block business logic]</i> Block either user from logging in.</p> <p>Resolution taken: Add <i>Unique</i> constraint to database.</p>
(a)	(b)
<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> <p>Violated constraint: /*Django-oscar*/ <i>Order.basket_id</i> is an integer-field rather than a <i>foreign key</i> to <i>Basket</i>.</p> <p>Consequence: <i>[Data corruption]</i> Potential data corruption and performance hits for user requests.</p> <p>Resolution taken: Add <i>Foreign key</i> constraint to database.</p> </div>	
(c)	

Figure 2.1: Three real-world issues [136, 168, 63] that violated three types of DB constraints and led to severe consequences. These issues are from popular open-source web applications. To fix the issues, the developers added the constraints to the database [137, 167, 61].

three common database constraints in recent years [129, 55, 156], enabling applications to easily specify and enforce constraints in databases.

However, despite these initiatives, many app developers still do not take full advantage of database constraints to protect their application data integrity against application bugs or operator errors. We observe in our study (§ 2.2) that many constraints (10-72) for each studied app were added as *afterthoughts*, i.e., they were missed first when columns were created and added much later, often because a data integrity issue was detected and resulted in damages.

There are many reasons for such negligence. As the industry demands more engineers to build various applications, many of them do not have solid database training and may not be aware of data integrity or constraints at all [144, 145]; Additionally, to err is human, even experienced developers can easily forget some required constraints due to deadline pressure.

2.1.2 Consequences of Missing Constraints

Missing DB constraints can result in severe consequences. Figure 2.1 shows three real-world examples [136, 168, 63] from three widely-used e-commerce and team chat applications. These issues were caused by inconsistent data stored in databases that violate the *not-null*, *unique*, or *foreign key* constraint. As a result, the applications suffer from severe consequences, such as page crashes and failed login attempts. For e-commerce, any such issue can lead to significant business loss [136].

To fix the problem caused by missing DB constraints, and more importantly *to avoid similar issues in the future*, developers added the missing data constraints into their corresponding databases [137, 167, 61]. Had these constraints been specified earlier, such issues would have been detected and reported before invalid data being inserted into databases in the first place and avoid the impact on users.

Missing DB constraints has two primary consequences:

- Without database constraints to guard data integrity, data corruption caused by application bugs can easily stay dormant for a long time before being exposed, impacting users and leading to business loss. Without early detection, the culprit application bug can cause many corruptions, making database repairing a more challenging task.
- Missing database constraints also introduces challenges in diagnosing such issues because it is difficult to trace back and identify *when* and *how* such inconsistent or erroneous data were added into the databases.

We can look into one of the examples [136] in Figure 2.1(a) from the popular e-commerce Saleor [138]. The app developers noticed a page crash caused by “an invalid order in database with a null total price.” However, they got stuck in identifying the root cause of the *null* record. After rounds of investigations in nine days, three developers finally found the application bug. To detect future similar application bugs earlier before they corrupt databases, developers added

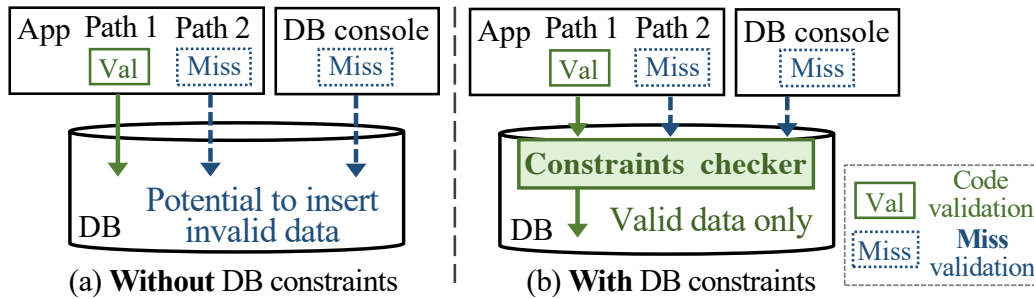


Figure 2.2: In (a) when the constraints are not enforced in database, missing validation in any code path or DB console could potentially cause invalid data to be inserted. Contrarily, in (b), when database constraints are enforced, even if validations are missed in some paths, database always conducts integrity checks and blocks invalid data as the final guard.

the *not-null* constraints into the database to “prevent reported weird and hard to reproduce bugs”, according to their commit comment.

2.1.3 Why DB Constraints Are Better Guards?

Interestingly, many developers think that their own application code can check against data integrity violations, and thereby there is no need to add DB constraints [80, 49]. Such assumptions often fail to protect data integrity in practice because there are multiple places that can change the database data and result in data integrity violations if not checked properly.

Specifically, as depicted in Figure 2.2, database data can be added or altered in various places throughout the application’s code, and some of them may not even be in the same piece of software (e.g. some batch job scripts to insert or change data in bulk). To make things even worse, software may implement some code logic in a different language [102], or by different teams. The fast turnover rates of today’s software engineers in IT companies further make it difficult to ensure that every single code location has proper integrity checks.

Figure 2.3 shows one such real-world example [62] from Oscar [118]. In this e-commerce application, each user’s email field needs to be *unique* as it is used for authentication. To ensure this, when a new user signed up, the application code checked whether that email already existed in DB. Unfortunately, on another code path that performed email updating for registered users,

```

/* django-oscar/apps/customer/forms.py */
class EmailUserCreationForm(Form): # Code path 1
    def save(self, email):
        if User.objects.filter(email=email).exists():
            raise forms.ValidationError("A user with
                that email already exists.")
        user.save()
        ↘ Validate uniqueness before save

-----
class UserAndProfileForm(Form): # Code path 2
    def save(self, email):
        user.save()
        → Miss validation → Save invalid data

```

Figure 2.3: A real-world issue [62] from Oscar caused by missing code validations. The user’s email should be unique. Developers specified the check in one code path (when new users registered) but forgot to check in another code path (when a registered user updated his email). To fix it, they enforced the *unique* constraint in DB.

there was no check at all. As a result, this application bug allowed the same email addresses to be used for two or more user accounts, causing many login issues. It took developers quite some time to diagnose, and even much longer to repair the database (since they needed to inform the affected users to change to another email address).

Moreover, application code checks for data integrity often fail during concurrent executions because of data races [160]. For example, two concurrently handled requests can both receive non-existent results from the data integrity check in the first query, and then both insert the same values in the second query, which violates the unique constraint.

A study on Rails applications [42] reveals that 13% of code validations for uniqueness and foreign key are error-prone during concurrent executions, as is also warned in web frameworks’ documentations [59, 130]. Our study in §2.2 also confirmed such observation. Even encapsulating validation logic within a transaction may not work because most production databases default to non-serializable isolation [42, 160].

Furthermore, DB admins can also manipulate data using the “backdoor”, i.e., the DB console, which bypasses all checks in the code. In comparison, in Figure 2.2(b), when constraints

are enforced by the DB, even if validations are missed in some paths, the DB always acts as the final guard to perform integrity checks and detect violations against specified constraints.

As such, we believe that web applications should *take full advantage of database constraints to ensure data integrity* when possible.

2.1.4 Our Contributions

This paper focuses on the problem of missing database constraints in widely-used web applications that leads to data integrity issues and results in system downtime and business loss.

First, we make one of the first attempts in understanding and evaluating the reality of the adoption of database constraints in today’s web applications. We study five popular web apps in Table 2.1 ranging from e-commerce to communication tools. Our study reveals several interesting findings: (1) Many (10-72) database constraints were missed in the beginning and were added much later as afterthoughts after some issues occurred. (2) Most (82%) of these cases could result in consequences, including page crash and data corruption of order-related or payment-related records. (3) Most (87%) issues that missed DB constraints also missed code checks in application code, indicating that solely relying on application code checks instead of leveraging database constraints is not a safe approach to guarantee data integrity (More details in §2.2).

Second, we leverage a unique observation that application code usually contains “hints” that imply certain data constraint assumptions made by developers. Figure 2.4 shows two examples of such code snippets. In (a), the code uses the column `col` as an identifier to check its existence, and only creates a new record if it does not already exist, indicating that the `col` is a unique identifier. In (b), the code invokes a method on `col`, indicating that `col` cannot be null. §2.3.3 shows all our discovered code patterns that imply data constraints.

By leveraging this observation, we build CFINDER which employs program analysis to analyze application source code to automatically infer and detect any missing database constraints to improve database integrity (against application bugs and operator mistakes).

```
if not Table.get(col=val).exist():  
    Table(...).save()
```

(a) Table (col) **Unique**

```
Table.col.method()
```

(b) Table (col) **Not null**

Figure 2.4: Code snippets with implicit assumptions on database constraints. (a) Unique constraint: Save record only when no record filtered by the column exists. (b) Not-null constraint: Invoke methods on the column which should not be nullable.

We evaluate CFINDER with eight widely deployed web applications, including an industry-strength software from a **commercial company with millions of users**. CFINDER has detected 210 missing DB constraints from these applications. We have reported **92 of them to the developers of these applications, so far 75 have been confirmed by these software**. The tool effectively detects the missing constraints with a precision of 78% for newly detected constraints and a recall of 79% for an existing dataset.

2.2 Understanding Missing Database Constraints in Web Applications

Before we build a tool to infer the missing constraints, we first aim to understand more about the current status of DB constraints in web applications. Specifically, we aim to answer: (1) *Is it common for developers to miss specifying some DB constraints?* We define “missing” constraints as those that are not specified when the columns are created, and added *later* in another pull request. Missing constraints indicate the potential vulnerabilities which allow invalid data to get stored in the database. (2) *Do these missing DB constraints lead to issues with severe consequences?* Finally, (3) *Do these missing constraints have validation checks in the code and whether the validations can protect the data integrity effectively?*

As a lens to answer these questions, we conduct the study on five widely-deployed real-world web applications listed in Table 2.1, representing app domains including e-commerce, team chat, etc. The apps are built on top of Django [57], a popular framework powering more than

Table 2.1: The web applications used in our study. *Stars*: Number of stars on Github. *LoC*: Lines of code.

App.	Category	Stars	LoC	#Table	#Column
Oscar [118]	E-commerce	5.2K	74K	77	773
Saleor [138]	E-commerce	15.3K	298K	98	1013
Shuup [141]	E-commerce	1.8K	196K	227	2236
Zulip [173]	Team chat	15.3K	361K	97	826
Wagtail [157]	Content management	11.7K	181K	60	841

94K web apps, including large commercial companies like Instagram [68].

To collect the history of adding DB constraints, we leverage the database migration files [60], which maintain the historical modifications to the database schema. From them, we collect the SQLs that add the new database constraints. To get the “*missing*” constraints, we further filter out the constraints that are added together with the creation of columns. To collect the related issues, we search the issue tickets that reference the commit of migration files. We then manually examine the issues to understand the root causes and severity based on developer comments and issue labels.

Threats to Validity The five apps in our study are specific to Python-based web applications using Django, which may not represent all web applications; Other web frameworks, like Rails (Ruby) and Hibernate (Java), let developers specify and use database constraints with similar primitives.

Observation 1: *Many constraints were added as afterthoughts, with 10-72 constraints missed first and added in later pull requests for each application (Table 2.2).* Such an overlook makes the studied applications vulnerable to invalid data, as it can potentially be stored in the database before the constraints are enforced correctly.

Observation 2: *A majority (82%) of these missing constraints were noticed and added by developers after data integrity issues were detected (Table 2.3). These issues could lead to severe consequences. Moreover, they took a long time (on average 19 months) to get exposed.*

Table 2.2: The number of database constraints that are missed first and added in later pull requests in each application.

App.	Oscar	Saleor	Shuup	Zulip	Wagtail	Total
Unique	22	10	5	16	6	59
Not-null	48	9	6	9	4	76
Foreign key	2	2	0	4	0	8
Total	72	21	11	29	10	143

We classify how developers find such missing constraints into four categories: (1) Developers were notified from 30 issue tickets for 31 (22%) missing constraints. Users were likely to have experienced some real-world issues with consequences. (2) After fixing the reported issues, developers sometimes realized that more data fields had similar issues. Thus, they added 59 (41%) such missing constraints. (3) The other 27 missing constraints belong to “Fixed by dev”, meaning that developers mentioned “fix”, “prevent issue”, etc. in comments, which indicates their purpose to fix an issue. (4) 22 (15%) were added due to new features or code refactoring.

We find that 30 different issues with detailed user reports have led to various severe consequences. Among them, 18 issues caused crashes, with 7 of them blocking critical business logic (order or payment-related for e-commerce), causing poor user experience and revenue loss. The other 8 caused data corruption, including order data and other users’ account data.

To make things worse, these missing constraints took a long time (on average 19 months) to be noticed and fixed, opening up a long vulnerable time window that allowed constraint-violating data to be inserted into the database.

Observation 3: *Most (87%) issues that missed database constraints also missed some required checks in the application code. For the rest (13%), even with code checks, the constraint-violating data was still stored during concurrent requests.* It indicates that the code checks are incomplete and insufficient. The 30 issues belong to three categories. (1) 22 (73%) have no checks at all in the application code. (2) Four (13%) issues have checks in some code paths but miss checks in other paths that manipulate the same data. It indicates that developers usually fail to ensure

Table 2.3: Reasons why developers add the *missing* constraints. The majority (82%) originates from issues, either from user reports or developers’ findings.

Type	Related to issue			Feature / Refactor	Unknown
	From reported issue	Learn from similar issue	Fixed by dev		
Unique	17	16	15	8	3
Not-null	11	40	12	12	1
FK	3	3	0	2	0
Total	31 (22%)	59 (41%)	27 (19%)	22 (15%)	4 (3%)
	117 (82%)				

multiple places adhere to the same constraints. (3) Interestingly, for the rest four (13%) issues that have full code checks, constraint-violating data still makes its way into the database. Developers suspected the reason was that code checks failed to handle concurrent requests [166]. They commented, “*This is clearly the result of a race, since we have this check in the view code*”, after careful diagnosis.

Implication In summary, even for these widely deployed web applications, database constraints are not fully leveraged by developers to protect their application data. A large number of database constraints are missing, causing issues with severe consequences. Moreover, the validations in the application code are ad-hoc and generally error-prone to concurrent requests, which makes the situation even worse.

2.3 Design and implementation

2.3.1 Design Choices: Possible Ways to Find Missing Constraints

Given the consequences brought by missing database constraints, the current practice of adding them after issues have been exposed is far from satisfaction. There are three possible approaches to identify the missing constraints:

Manual inspection Letting developers inspect the whole database schema manually requires

their expertise in both database and business logic. It is tedious and error-prone even for domain experts, considering the large number of tables (up to thousands) and columns (up to hundreds per table).

Infer from production data Another approach is to discover from the production data. For example, if a column has a predominant percentage of records that satisfy a certain constraint (e.g., 99.99% are not-null), a potential constraint is indicated.

Though the idea is intuitive, it has three main limitations. (1) Approaches based on statistics are usually biased and limited by insufficient datasets. E.g., some rare cases may allow the insertions of null data, but the cases have not been triggered yet. As a result, the wrong conclusion of a not-null constraint can be drawn from the data. Similarly, due to lack of data, the idea does not apply to newly created tables or added columns, from which most of the missing constraints originate. (2) It is cumbersome for developers to gain access to the production data, especially with access control and privacy concerns. (3) This approach has unacceptably high false positive rates [35, 36]. In previous work [45], 95% of discovered statistically-valid unique constraints are false positives (see more details in §2.5).

Infer from application code Inferring constraints from the code logic has several advantages. Compared to data, the source code (1) is not limited by data and (2) contains the business logic of what constraints the data should follow in semantics. Moreover, developers can always cross-check the inferred constraints with the production data. The major concern is, given the code complexity and diversity, *how many data constraints can possibly be inferred from the source code?*

After looking into several real-world web applications, to our surprise, we observe many code patterns that have implicit assumptions on database constraints for all three constraint types. Developers have the assumptions about data constraints in mind, thus their code implementation that retrieves or manipulates the data will follow certain patterns. We list the observed patterns for each constraint type in Figure 2.6 (§2.3.3).

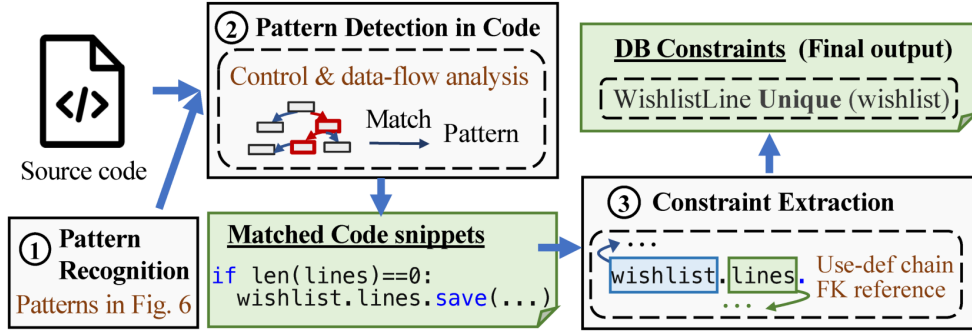


Figure 2.5: The overview of CFINDER. CFINDER contains three steps to infer the missing database constraints from the application source code. The green boxes are the output of the steps.

Based on the above trade-offs and observations, we choose to extract missing database constraints from the application code.

2.3.2 CFINDER Overview

Figure 2.5 illustrates the three steps of our approach. In step ①, we recognize the code patterns that imply certain DB constraint assumptions (§2.3.3). In step ②, with observed patterns and application code as input, CFINDER applies control and data flow analysis to find code snippets that match each pattern’s conditions (§2.3.4); In step ③, from the found snippets, CFINDER extracts and infers the formal DB constraints (§2.3.5); The output of CFINDER is the set of missing DB constraints.

The static analysis is flow-sensitive. It is also field-sensitive because CFINDER treats the fields of a model class differently. Currently, it does not consider alias. In our evaluation, we didn’t catch any false positives caused by aliasing.

2.3.3 Code Patterns with Assumptions on DB Constraints

Code Patterns

From many web applications, we have observed that many code patterns with assumptions on each constraint type widely exist, but have not been studied before. Figure 2.6 lists the patterns we discovered, along with real-world examples from e-commerce apps. We name each pattern as $PA_{(type)(idx)}$, where *type* stands for constraint types and *idx* is the index.

Check existence before save/error handling (PA_{u1} *unique*): The code explicitly checks if the data constraints hold. As Figure 2.6a shows, it first retrieves records filtered by `product`, then only saves a new record if no existing record returns. It reflects developers' intention on uniqueness: only one record with the value of `product` can exist in DB. Similarly, the pattern can be extended to do error-handling after the check, i.e., throwing exceptions when the record already exists.

APIs with assumptions (PA_{u2} *unique*): Web frameworks provide developers with `QuerySet` APIs [58] to encapsulate data manipulations. Some APIs are implemented with similar assumptions as *Check existence before error handling*. For example, `get` uses `column(s)` as the unique identifier to retrieve the record and throws an exception when multiple records are returned [56]. Thus, when developers use this API, they expect the `column(s)` to be unique. Such APIs include `{get, get_or_create, get_obj_or_404}` in Django.

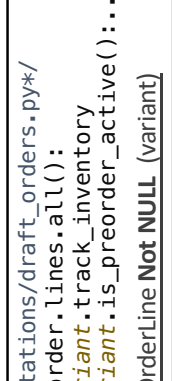

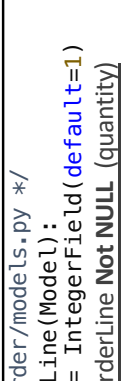
Method/field invocation on column without NULL check (PA_{n1} *not-null*): When invoking a method or accessing a field on a column, the column should be not-null. Otherwise, invocation on NULL will throw an exception. We further exclude cases that have explicit NULL checks before the invocation, as the check avoids the exception, making them false positives.

Check NULL before assignment/error-handling (PA_{n2} *not-null*): Similar assumptions as PA_{u1} can be applied to *not-null* with some tweaking. For example, when `order.creator` is null, the app raises an error "Anonymous orders not allowed". One variant is, when the field is NULL, the code explicitly assigns a value to the field before saving, making it not-null.

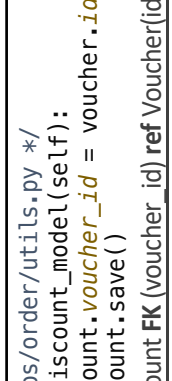
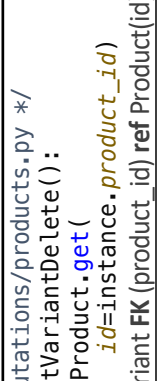
Pattern with Assumptions	Control flow graph	Real-world Code Example	Explanation on assumption
PA_{u1} for Unique: Check existence before save/error-handling		<pre> /* Oscar: wishlists/models.py */ lines = wishlist.lines. filter(product=product) if len(lines) == 0: wishlist.lines.save(...) Implies: <u>WishlistLine Unique (product,wishlist)</u> </pre>	Only save record when no record filtered by the column(s) exist.
PA_{u2} for Unique: APIs implemented with assumptions		<pre> /* Oscar: wishlists/views.py */ if to_wishlist.lines. filter(product=product).count() > 0: raise Error("Wishlist already containing product") Implies: <u>WishlistLine Unique (product,wishlist)</u> </pre>	If one record filtered by the column(s) already exist in database, throws an exception.
PA_{u2} for Unique: APIs implemented with assumptions		<pre> /* Oscar: dashboard/orders/views.py*/ order = Order.objects.get(number=request.GET['order_number']) Implies: <u>Order Unique (number)</u> </pre>	Use column(s) as the unique identifier to retrieve data: if more than one results exist, throws an exception.

(a) Patterns with assumptions on *Unique* Constraint.

Figure 2.6: Code patterns with implicit assumptions on three DB constraint types, together with real-world examples and explanations.

<p>PA_{n1} for Not-null: Method/field invocation without NULL check</p>		<pre>/* Saleor: mutations/draft_orders.py*/ for line in order.lines.all(): if line.variant.track_inventory or line.variant.is_preorder_active():... Implies: <u>OrderLine Not NULL</u> (variant)</pre>	<p>Invocation on a column is only valid when the column is <i>not null</i>. Having NULL check before makes it a false positive.</p>
<p>PA_{n2} for Not-null: Check NULL before assignment /error-handling</p>		<pre>/* Shuup: models/orders.py*/ class Order(Model): if not self.creator: raise Error("Anonymous orders not allowed.") Implies: <u>Order Not NULL</u> (creator)</pre>	<p>If the column value is NULL, throws an exception or explicitly assign it a value.</p>
<p>PA_{n3} for Not-null: Field with default value</p>		<pre>/* Oscar: order/models.py */ class OrderLine(Model): quantity = IntegerField(default=1) Implies: <u>OrderLine Not NULL</u> (quantity)</pre>	<p>The column has a default value assigned and no place sets the column to be null.</p>

(b) Patterns with assumptions on *Not-null* Constraint.

<p>PA_{f1} for FK: Column referring primary key</p>		<pre>/* Oscar: apps/order/utills.py */ def create_discount_model(self): order_discount.voucher_id = voucher.id order_discount.save() Implies: <u>Discount FK</u> (voucher_id) ref Voucher(id)</pre>	<p>Dependent table's column is assigned the value of Referenced table's primary key.</p>
<p>PA_{f2} for FK: Primary key referring column</p>		<pre>/* Saleor: mutations/products.py */ class ProductVariantDelete(): product = Product.get(id=instance.product_id) Implies: <u>Variant FK</u> (product_id) ref Product(id)</pre>	<p>Use Dependent table's column value when filter by Referenced table's primary key.</p>

(c) Patterns with assumptions on *Foreign Key* Constraint

Figure 2.6: Code patterns with implicit assumptions on three DB constraint types, together with real-world examples and explanations. (cont.)

Field with default value (PA_{n3} *not-null*): Some fields have a default value, which works similarly as PA_{n2} , i.e., assigns the default value to the field if it has not been set before saving. If nowhere in the code would explicitly assign the field a null value, we assume it is not-null.

Column referring primary key (PA_{f1}, PA_{f2} *foreign key*): Patterns in Figure 2.6c reflect the *referential* assumption between tables: the column in the dependent table refers to the primary key (PK) value in the referenced table (PA_{f1}), or vice versa (PA_{f2}). For example, in PA_{f1} , the value from `Voucher` (referenced table)’s PK is saved to `Discount` (dependant table)’s column named `voucher_id`, indicating that `Discount.voucher_id` should be a foreign key to `Voucher`.

Our evaluation in §2.4.2 and §2.4.3 shows that these code patterns are effective for detecting missing DB constraints (found 210 previously unknown constraints) and have good coverage (79% recall on a collected dataset). We also discuss potential improvements and extensions to the patterns there. Besides, since these patterns reflect semantic code logic, they are general and applicable to applications in other frameworks or languages.

Conditions of Code Patterns

After observing these code patterns, a natural question would be how to detect them in the application code. A naïve way is to represent the patterns with some predefined regular expressions and match the code with them. This may work for simple cases with well-defined APIs, such as `get`. But it cannot detect most other cases. Take PA_{n2} “check NULL before assignment” as an example, matching any assignment after any NULL check would introduce too many false positives, since the two operations could come from unrelated code blocks and operate on unrelated data. Such complex control and data logic can hardly be defined and matched with regular expressions. Moreover, it cannot infer the table of the constraints as that requires the data flow information (§2.3.5).

Instead, CFINDER represents patterns as the conjunction of three types of conditions, which involve control and data dependencies built on top of the abstract syntax tree (AST) [128].

Based on it, our detection algorithm (§2.3.4) traverses the AST and finds snippets that match all conditions of a pattern.

To introduce the three types of conditions, we use the first pattern PA_{u1} for unique constraint in Figure 2.6 as the example, which *checks existence before save/error handling*.

Control dependencies (C-D) Each pattern consists of several sub-components (subtrees in AST), and each subtree has its specific semantic meaning. These subtrees follow certain control dependencies. For example, PA_{u1} requires two sub-components, *check existence* and *save*. They represent two subtrees that satisfy the control flow of the *IF* block, i.e., *condition* for *check existence*, and *body* or *else* for *save*.

Other types of control dependencies include one syntax tree T_i being the parent of another tree T_j , etc. Using another pattern PA_{n1} as the example, we require that for all parent trees of the *field invocation*, no one T has a condition branch T_{cond} that has the NULL check.

Syntax pattern matching (P-M) As we mentioned, each subtree needs to represent a specific semantic meaning. To bridge their gap, we *pre-define* a set of syntax-based patterns P_* , where each P_* consists of a category of simple syntax tree patterns with the same semantic meaning S_* . Therefore, whether a subtree T_* represents a semantic meaning S_* can be evaluated by T_* matching with one syntax pattern of P_* .

For example, we define P_{exist} to represent the category of patterns indicating a check on the existence of a record. One such syntax tree could be a `Call` block with a `Attribute` subtree with name `exist` (Check more in Figure 2.7). These syntax patterns are general to the framework and easy to customize.

Back to PA_{u1} , it requires: (1) the if-condition checks whether the record *exist* or *not exist*, i.e., T_{cond} matches with P_{exist} or $\neg P_{exist}$. (2) Respectively, the two subtrees T_{body} and T_{else} in two branches match with P_{save} (save record when not exist) or P_{error} (error-handling when a record exists). The results (R_*) of these syntax pattern matching are connected with AND and OR, to form the final evaluation of this condition.

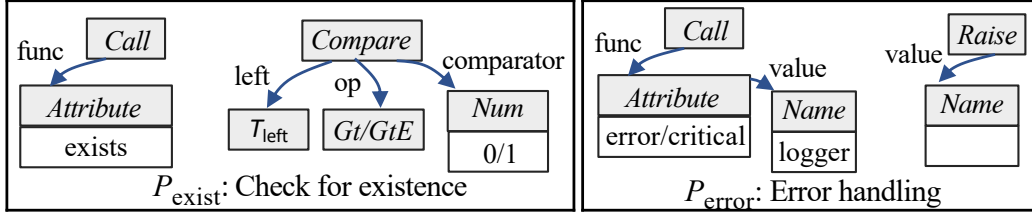


Figure 2.7: Example of pre-defined syntax tree patterns. We use them to match with the candidate syntax trees. Each category of P_* can have several patterns representing the same semantic meaning.

Data dependencies (D-D) This condition requires the data in subtrees to follow certain data dependencies, i.e., the two subtrees operate on the same tables and columns.

In PA_{u1} , we require the match of the table and column that (1) get saved in T_{body} and (2) perform the NULL check in T_{cond} . We evaluate the data dependencies by first inferring those tables and columns from each subtree using data-flow analysis (§2.3.5) and then matching them.

To sum up, we list the formal representation of PA_{u1} :

$$(C - D) [T_{cond}, T_{body}, T_{else}] = \text{IF_block_subtrees}()$$

$$(P - M) R_{cond} \wedge (R_{body} \vee R_{else}), \text{ where}$$

$$[R_{cond}, R_{body}, R_{else}] =$$

$$\text{MATCH}([T_{cond}, T_{body}, T_{else}], [P_{exist}, P_{error}, P_{save}]) \vee$$

$$\text{MATCH}([T_{cond}, T_{body}, T_{else}], [\neg P_{exist}, P_{save}, P_{error}])$$

$$(D - D) \text{DataDepend}(T_{cond}, T_{body} \vee T_{else})$$

2.3.4 Code Patterns Detection Algorithm

In step ②, CFINDER detects code snippets that can match the conditions of one code pattern from the application code. Taking the first code snippet for PA_{u1} in Figure 2.6a as the example, we show how it can be detected from the code.

Overall Algorithm

The steps are as follows.

- CFINDER walks the module’s AST in a breadth-first fashion to identify the candidate code snippets whose root types match the pattern’s root type (IF node in PA_{u1}).
- For each code snippet, CFINDER then extracts their subtrees following the *control dependency* of the pattern, i.e., extracts subtrees $T_{\text{cond}}, T_{\text{body}}, T_{\text{else}}$ from the root IF node.
- CFINDER then performs the *syntax pattern matching* on each subtree. E.g., match subtree T_{body} (`wishlist.lines .save`) with predefined P_{save} (details in next paragraph).
- CFINDER further checks the *data dependencies* using the use-definition graph to see if variables in two subtrees refer to the same table and columns (details in §2.3.5).
- If all pattern conditions evaluate to True, then we find a candidate snippet with assumptions on DB constraint.

Match Subtree with Syntax Pattern

Figure 2.8 shows the syntax tree of the example snippet on the left, with some subtrees collapsed. The MATCH function matches its subtree T_{body} (left) with the predefined syntax pattern P_{save} (right).

Here, P_{save} represents the category of syntax patterns that have the meaning of “saving a record”. In the AST form, one example of the syntax pattern is a Call node calling an Attribute node named `save` or `create`.

To implement MATCH, CFINDER performs a breath-first traversal in T_{body} and finds the node which matches the root of P_{save} , i.e., the Call node. Then for each child node of Call in P_{save} (the Attribute node), CFINDER checks if there is a corresponding subtree node in T_{body} .

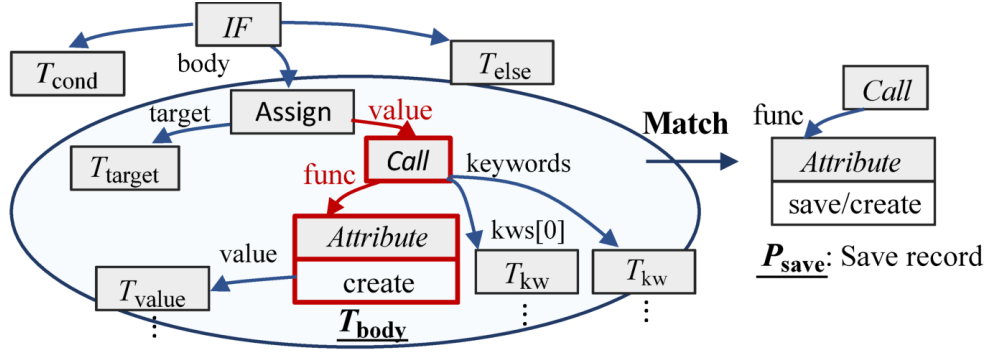


Figure 2.8: Matching syntax tree T (left) with pre-defined syntax pattern P (right). The subtree with bold red borders on the left is the match. T (left) is constructed from the example in Figure 2.6a.

CFINDER recursively repeats this process until the leaf nodes of P . If all children have a match, CFINDER concludes that T_{body} matches P_{save} .

Figure 2.7 shows more examples: two categories of pre-defined syntax patterns for P_{exist} and P_{error} . Note that these patterns are as simple as a syntax tree with a depth of only one or two, and they have no control or data dependencies. We collect them heuristically by studying the application code. They are general to applications, and *more importantly*, they can be easily customized and extended.

2.3.5 Database Constraints Extraction

In step ③, CFINDER automatically converts the snippets into formal DB constraints. After detecting the second code snippet in Figure 2.6a that matches PA_{u1} , in Figure 2.9, this step infers the table `WishListLine` and columns `(wishlist, product)` from it. To achieve it, CFINDER traces the definitions in code using use-definition analysis [81] and table metadata.

Identify the Table

The MATCH step identifies the variable list `(to_wishlist.lines)` that represents the table object. However, identifying which table it represents (`WishListLine`) is often *non-trivial* due

to two challenges:

- Python’s language feature, dynamic typing, i.e., a variable’s type is defined at runtime. Therefore, static analysis doesn’t know `to_wishlist`’s type (class of `WishList` table).
- The “variable” may involve a chain of field accesses, which transfers from one table to another following the foreign key reference. For a real example in Oscar, `self.attribute.option_group.options` involves the reference between three tables. It is hard to sort out the relationship with such complex code even with human inspection.

To handle the first challenge, CFINDER *infers the definition using use-def chain analysis*. Starting from the first variable, CFINDER traces its definitions in the use-definition chain and identifies one of the definitions being table class. In the example, `to_wishlist` gets the definition from `WishList.objects.get`, which returns an instance of `WishList` class. To be scalable to large applications, CFINDER does not perform the inter-procedure analysis.

Second, CFINDER follows the list of field accesses and tracks the corresponding tables using the table’s metadata. Starting from `to_wishlist`, which is an instance of `WishList` class, `.lines` retrieves the instance of a `WishListLine` class through the foreign key reference. CFINDER repeats this process until the end of the field list.

Identify the Column

The columns of the not-null and foreign key constraints are usually obvious and CFINDER gets them directly from the specified patterns. Here we discuss two special cases, i.e., *composite* and *conditional* unique constraints:

- When retrieving referenced objects through the foreign key field, it contains the *implicit join* on table ID. In the example, `to_wishlist.lines` retrieves the lines related to the `to_wishlist` instance. Consequently, besides `product`, the generated SQL statement filters on `wishlist_id`


```

/* Oscar: wishlists/views.py */
class MoveProductToAnotherWishList(View):
    def get(self, request):
        to_wishlist= WishList.objects.get(key=xxx)
        if to_wishlist.lines.filter(product=xxx).count()>0:
            raise Error("WishList already containing product")
class WishList(Model):
    lines = ManyToOneRel(WistListLine)

```

Inferred constraint: WishListLine Unique (wishlist, product)

Figure 2.9: Infer the constraint table from code: (1) Infer the definition using data flow analysis (2) Track the tables along the chain of field accesses. Specifically, `to_wishlist` represents the `WishList` class and `to_wishlist.lines` refers to the final `WishListLine` class through the foreign key.

as well. Thus, CFINDER infers that the final constraint requires columns (`wishlist, product`) to be *composite unique*.

- When retrieving records by filtering on columns with fixed values (e.g., `filter(col, valid=True)`), it indicates a “partial (conditional) unique constraint” [126], which restricts the uniqueness of `col` over a subset of data defined by the condition (`valid=True`).

Get Missing DB Constraints

After inferring all DB constraints from the code, CFINDER filters the existing constraints retrieved from `information_schema` tables of databases.

2.4 Evaluation

As shown in Table 2.4, we evaluate CFINDER on eight large web applications including seven widely-adopted open source web applications and the main web application of one commercial enterprise (COMPANY) with millions of end-users. The open-source applications are top-starred in each category on Github, with three of them having 10K stars and five having

5K stars. Moreover, Saleor is adopted by e-commerce companies including one with 50M revenue [139], Edx by 160 institutes and has millions of users [66], Zulip by large communities and universities [169], etc. These open-source applications have 74K to 617K LOC, more than 60,000 commits, and have high demands on data integrity and reliability due to their wide adoption and millions of users. We use the latest version of all applications (commit hashes are in the references).

We evaluate the effectiveness of CFINDER based on how many new *missing* database constraints can be detected (§2.4.1). We further report them to the app developers and get their confirmations (Table 2.4).

Moreover, we evaluate the precision of the detected missing constraints (§2.4.2) and study the reasons for false positives. We have two human inspectors independently examine the detected missing constraints and label a case as true positive only when consensus was reached. Furthermore, we evaluate the coverage (recall) of CFINDER (§2.4.3) on two datasets. The first dataset contains all the existing (not missing) database constraints already set by the latest application code. The second dataset contains 117 real-world *missing* constraints collected from the past commit history (Table 2.3). These missing constraints were noticed because data integrity issues were detected. We further evaluate CFINDER’s performance (§2.4.4) and discuss the developer’s feedback (§2.4.5).

Databases are fully set up and populated with testing data only. All experiments are done on a single machine with a 2.30GHz CPU (6 core), 16GB Memory and 256GB SSD running a Ubuntu 18.04 distribution.

Table 2.4: Evaluated applications and detected missing DB constraints from them. “*Detected existing*”: Detected constraints that already exist in DB. “*Detected missing*”: Detected constraints that miss in DB. “*ACK by dev*”: numbers of missing constraints acknowledged by developers that need to be added. For three apps with zero confirms, we received no response to our issue reports.

App.	Category	Github stars	LOC	Detected existing	Detected missing	ACK by dev
Oscar [118]	E-comm	5.2K	74K	159	24	5
Saleor [138]	E-comm	15.3K	298K	220	15	0
Shuup [141]	E-comm	1.8K	196K	290	31	0
Zulip [173]	Team chat	15.3K	361K	265	21	12
Wagtail [157]	CMS	11.7K	181K	69	10	7
Edx [116]	Online course	6K	617K	509	43	0
EdxComm [67]	E-comm	122	93K	97	14	6
COMPANY	Enterprise	-	-	-	52	45
Total	-	-	-	1609	210	75

2.4.1 Effectiveness in Detecting *Missing* DB Constraints

Overall Results

Table 2.4 shows the number of detected missing database constraints from each web application. Overall, CFINDER detects 210 missing database constraints from eight web applications, including 10-43 missing constraints for each open-source web application and 52 missing constraints for a commercial company with millions of users.

We manually validated the detected constraints and reported the identified true missing constraints to app developers. When we contacted the developers, we prioritized these applications that actively responded to our issue reports. For three apps with zero confirms, we received no response to our reports.

So far we reported 92 of them and we have got **75** confirmed by developers as real missing database constraints, including 30 of them from seven open-source web applications and 45 from the commercial company. Among the 75 confirmed constraints, there are 37 unique constraints,

Table 2.5: Examples of confirmed missing database constraints. The first two examples are already merged in their main branches.

Confirmed Unique constraints (37 cases)	
Example:	ProductAttr Unique(code,product_class) [64]
Potential consequence:	Product attributes with same attribute code for a product class are invalid and invisible to customers.
Confirmed Not null constraints (22 cases)	
Example:	Attachment Not NULL (realm) [170]
Potential consequence:	The attachment is not valid when uploaded without a realm (organization). Similar as a data loss to users.
Confirmed Foreign Key constraints (16 cases)	
Example:	OrderDiscount (offer) Ref Offer (id) [65]
Potential consequence:	The discount on an order is not valid without linking to an existing offer.

22 not-null constraints, and 16 foreign key constraints. We provided one example for each constraint type in Table 2.5 to demonstrate the potential consequence of not having the missing constraints.

Breakdown of the Detected Missing Constraints

To understand the effectiveness of CFINDER in detecting each type of *missing* database constraints, we present the breakdown for different code patterns in three constraints types, *Unique*, *Not-null*, and *Foreign key* in Table 2.6.

- *Unique constraint*: CFINDER detects 66 missing *unique* constraints, with two code patterns detecting 16 and 56 respectively. Moreover, among them, 13 are “partial unique constraints” (§2.3.5). Some app developers are not aware of this type of constraint, thus not taking advantage of them.
- *Not-null constraint*: For total 77 detected constraints, three patterns detect 44, 11, 22, respectively.
- *Foreign key constraint*: CFINDER detects 15 missing *foreign key* constraints in total. The

Table 2.6: The breakdowns of the number of detected *missing* database constraints for each constraint type and code pattern. One constraint can be detected by multiple code patterns, and we only count them once in *Tot.*(Total).

App.	Detected <i>missing</i> constraints									
	Unique			Not null				Foreign Key		
	PA _{u1}	PA _{u2}	Tot.	PA _{n1}	PA _{n2}	PA _{n3}	Tot.	PA _{f1}	PA _{f2}	Tot.
Oscar	3	10	12	9	1	0	10	1	1	2
Saleor	2	3	5	7	0	1	8	1	1	2
Shuup	2	4	6	12	5	7	24	1	0	1
Zulip	5	7	10	2	1	4	7	2	2	4
Wagtail	0	4	4	2	0	4	6	0	0	0
Edx	3	22	23	6	3	6	15	1	4	5
EdxComm	1	6	6	6	1	0	7	0	1	1
Total	16	56	66	44	11	22	77	6	9	15

number is relatively small, which is consistent with our study (§2.2) on real-world missing constraints in history. A possible reason is that when developers use the field to reference another table, the referential relationships are usually so obvious that developers are unlikely to neglect them.

2.4.2 False Positives in Detected Missing DB Constraints

As Table 2.7 shows, CFINDER’s precision in detected missing constraints is reasonably high for all three types of database constraints, 82%, 75%, 80% for unique, not-null, and foreign key constraints, respectively.

In total, 34 false positives (FPs) are introduced. There are two main reasons. *First*, 12 (35%) FPs are caused by the static analysis being unsound. Five have wrongly inferred database tables (§2.3.5) and seven have unrecognized or implicit NULL checks before the field invocation (thus these columns could be NULL without throwing exceptions). These FPs could be mitigated by fine-tuned code analysis, such as incorporating the inter-procedure information. *Second*, 13 (38%) FPs are caused when code matches the pattern but contains no assumption on constraints.

For example, one code snippet satisfies the pattern PA_{u1} , but it is only meant for a sanity check to handle a special case of no valid voucher, which does not involve uniqueness assumptions. To prune those FPs, CFINDER can further refine the patterns with finer-grind semantics.

Impacts of False Positives

The reported FPs can be easily recognized by developers, thus will not cause serious consequences. (1) Developers won't be misled after checking the code snippet (reported by CFINDER) that implies the constraint. For example, developers who read the error message can easily determine if it warns about a constraint violation. (2) Developers can run simple scripts to automatically check if the constraint is consistent with the production data, i.e., using data-driven approaches as complementary. (3) Even if they wrongly add a constraint, the DBMS will reject the schema migration if any existing data violates it. Developers then decide whether this is a FP or if data cleaning is required. In either case, if a constraint can be added, existing data must adhere to the constraint already.

Human Inspection Efforts

(1) It took two graduate students about 40 hours to manually inspect the FPs from 158 constraints that CFINDER reported in open source applications. Most time is spent on understanding how the field is used all over the codebase. (2) Based on our interactions with app developers, they are familiar with code and they do thorough inspections including the production data. The inspection time is acceptable. Half of the missing constraints we reported to Zulip's work channel are diagnosed within 20 minutes.

2.4.3 Coverage of Database Constraints

We then evaluate the percentage of database constraints that CFINDER can cover in its detection, i.e., the recall of CFINDER, on two different datasets. We further look into what are

Table 2.7: The precision of detected missing constraints by CFINDER. *Tot.*: Total number of detected missing DB constraints. *Precision*: $\frac{\text{TruePositive}}{\text{TruePositive}+\text{FalsePositive}}$.

App.	Unique			Not null			Foreign Key		
	Tot.	TP	Precision	Tot.	TP	Precision	Tot.	TP	Precision
Oscar	12	9	75%	10	8	80%	2	2	100%
Saleor	5	3	60%	8	7	88%	2	2	100%
Shuup	6	5	83%	24	17	71%	1	1	100%
Zulip	10	7	70%	7	5	71%	4	2	50%
Wagtail	4	4	100%	6	4	67%	0	0	-
Edx	23	20	87%	15	11	73%	5	4	80%
EdxComm	6	6	100%	7	6	86%	1	1	100%
Overall	66	54	82%	77	58	75%	15	12	80%

missed by CFINDER.

Evaluation with Existing DB Constraints

Even though the goal of CFINDER is to detect the *missing* constraints, we can evaluate whether the *existing* constraints behave consistently with the code patterns. Specifically, we evaluate how many *existing* DB constraints already set in the database can be covered by CFINDER. It reflects the generalization of the patterns. Note that we exclude foreign keys, as the existing ones are used differently from the patterns for missing ones. Specifically, for foreign keys that already exist in DB, developers mostly retrieve the referenced table through field invocations, such as `order.product` when `product` is a FK.

Table 2.8 shows that CFINDER has a reasonable recall. It can detect 61%-74% of unique constraints and 70%-83% of not-null constraints for seven web applications.

We randomly sample and study 40 false negatives for each of the two constraint types. They belong to three categories: (1) 57 (71%) do not exhibit any general patterns with assumptions on constraints. Among them, 20 are fields used for specific purposes and they might be improved by incorporating some application-specific domain knowledge. For example, some fields are used

Table 2.8: The percentage of *existing* constraints already set in the database that CFINDER can cover in the detection.

App.	# <i>Existing</i> constraints already set		What percentage can CFINDER cover	
	Unique	Not null	Unique	Not null
Oscar	49	156	67%	81%
Saleor	70	210	74%	80%
Shuup	89	298	70%	77%
Zulip	47	278	72%	83%
Wagtail	18	79	61%	73%
Edx	133	569	65%	74%
EdxComm	30	110	67%	70%

in the URL as the identifier, which may imply uniqueness. (2) 17 (21%) are fields not used in the application code logic, just placeholders for legacy or future use. (3) 6 (8%) have usages with assumptions but are not detected, mainly because code patterns are hard to recognize when they span different functions in the call chain. These can be improved by tracing the inter-procedure information.

Evaluation with Dataset on Missing Constraints

In our study (§2.2), we collect a dataset of 117 *missing* database constraints from the schema migration history (Table 2.3). These *missing* constraints were noticed after having some data integrity issues that caused real damage. We evaluate if CFINDER can detect these missing constraints on old versions of code, which could help *prevent the issues from happening*.

Table 2.9 shows that CFINDER has a good coverage. Out of the 117 real-world missing constraints in the dataset, CFINDER can detect 93 (79.5%) of them. These missing constraints would be caught if CFINDER had been adopted. We failed to detect 24 constraints mainly because they are too specific, i.e., do not exhibit general patterns. Note that we also mark those constraints that “learned from similar issues” as detected if the original issue is detected.

Table 2.9: The percentage of *missing* constraints that CFINDER can cover in the collected dataset. The dataset from our study (§2.2) contains constraints that missed first and added by later commits.

# <i>Missing</i> constraints in dataset			What percentage can CFINDER cover		
Unique	Not null	Foreign Key	Unique	Not null	Foreign Key
48	63	6	79%	83%	50%

Table 2.10: Time (seconds) to run CFINDER’s static analysis.

App.	Oscar	Saleor	Shuup	Zulip	Wagtail	Edx	EdxComm
Analysis time (s)	22	64	75	59	40	147	30

2.4.4 Performance of CFINDER

CFINDER is designed to run in the testing environment thus its performance is not time-critical. Table 2.10 shows that the analysis time of CFINDER’s static code analysis is less than 150 seconds for each application, and is near proportional to the application’s lines of code (up to 620K LOC for Edx).

2.4.5 Developers’ Feedback Discussion

We reported 92 of the detected constraints to the application developers and have got 75 confirmed so far. The others are rejected or still under investigation. Here we share the experience of the interactions with developers.

We are encouraged by the positive feedback from many developers of the evaluated applications. For example, Zulip developers quickly responded to our reported issues and actively examined their code base for similar issues with us [170, 172]. The confirmed missing constraints were either due to a lack of considerations in the design, or due to missing checks after business requirements changes. As one developer replied in the report for a not-null constraint, “*Being*

after that migration has run, ...,there's no reason to keep it nullable”.

In contrast, we find that maintainers hesitated to enforce some missing constraints we reported. For example, in one issue [119], the developers worried that the data migration might take too long a time to process the null values for large tables. In another issue, the developers assume that the invalid record will not be generated during normal workloads in current code logic, and thus are reluctant to add fixes [171].

2.5 Related Work

Empirical study of data constraints in web applications Previous studies have investigated the adoption of data constraints in the *application layer* [42, 164]. Bailis et al. [42] study the effectiveness of *application-level validations* as substitutes for their respective database constraints counterparts in web frameworks (Rails). Their quantitative experiment shows that app validations lead to data corruption due to concurrency errors in 13% of usages. Yang et al. [164] study the location, expression and evolution of data constraints. They find that developers struggle with maintaining *consistent* data constraints among the front-end browser, the application (using framework’s validation APIs), and the database. In contrast, our study in §2.2 focuses on the *missing* constraints neglected by developers in the database layer, which motivates tooling support to systematically detect the missing constraints.

Detecting data dependencies from applications Yang et al. [164] study the constraints specified in framework’s validation APIs and their inconsistencies with constraints in the database. Liu et al. [95] detect constraints specified in framework’s validation APIs in model classes with the motivation to use constraints to optimize query execution performance.

Our work differs largely in the following ways. (1) These works require developers to *already know and specify* these constraints using validations. In other words, they cannot help with the *missing database constraints neglected by developers*. Thus, our identified missing

constraints cannot be discovered by their works. Besides, in order to infer from the code logic with implications, CFINDER proposes more advanced code analysis algorithms. (2) Their goal is to optimize the performance or study inconsistencies, while CFINDER proposes to enforce the *missing* DB constraints to protect the data integrity. (3) Majority (88%) of their detected constraints are defined only in the framework level and are not DB built-in constraints, as they stated “defining inclusion and format constraints requires writing UDFs, which is tedious to implement in most DBMS” [95]. Thus they are orthogonal to CFINDER.

Inferring constraints from data Previous works on data profiling [35, 36] discover the data constraints by collecting statistics about the data itself. Aside from the limitation of biased and insufficient datasets we discussed in §2.3.1, these works still lack effective techniques to discover missing constraints that apps truly require. Specifically, as unique or foreign key constraints involve multiple columns, they traverse the search space of a powerset of column combinations and validate if the data satisfies the constraint. A majority of works focus on pruning the search space [37, 83, 45, 165, 120]. However, it is understudied which of the discovered *statistically-valid* constraints are truly required by apps in semantics. In fact, a vast majority (>95%) of them are false positives [45, 36]. Some [165, 134, 71] propose heuristic rules to prune FPs, but their effectiveness lack evaluations on real world large datasets.

In contrast, the source code (1) is not limited by data quality and (2) reflects what constraints the data really needs to follow in semantics. The evaluation shows CFINDER introduces reasonable precision (78%) and recall (79%).

Invariant detection from trace The line of work on invariant detection tools, like Daikon [69, 70], dynamically traces program runtime states and infers likely invariants in code. Typically dynamic approaches have a challenge of coverage problem. For likely invariants, the coverage problem of test cases or product runs can also lead to many false positives and false negatives, particularly false positives.

Application verification and synthesis using constraints. Another line of work focuses on

using data constraints for program verification and synthesis. Li et al. [93] detect the application bugs that violate the numerical data assertions inferred from the data. Wang et al. formally verify the equivalence of programs with different DB schemas [158] and synthesize equivalent programs [159]. These works are orthogonal and may help with code evolution when adding new constraints.

Leveraging constraints to improve performance and security Various constraints have been used to find better query plans and optimize query performance [94, 162, 95]. Our work reveals that there are opportunities to find more required database constraints, thus could complement their works.

Some works study methods to impose and verify the security and privacy “policies” [88, 109, 163]. These policies are usually too complex to be supported by current databases, thus are orthogonal to our work. Future work can study the automatic detection of these missing privacy-related policies from code. They are promising to improve the data quality in the further.

2.6 Limitation & Discussion

CFINDER targets on web applications that are backed by RDBMS and have a high requirement on data integrity, which widely exist in our daily life. Some systems shift the responsibility of data quality to the application layer as a design choice for better scalability and customization. It includes apps backed by NoSQL databases, which typically do not support constraints in DB. Though not our targets, CFINDER can still benefit them by identifying the missing data constraints and helping them check at the application/framework level. Moreover, NoSQL databases such as MongoDB recently start to support constraints at the database [105] level, showing its importance and potential.

CFINDER is currently implemented for Python-based web applications, as it relies on web frameworks’ APIs to identify database operations when performing pattern matching in §2.3.4.

For example, we use Django’s five APIs for record retrieval, three for record creation or updating, and one for existence check. However, CFINDER’s code patterns in §2.3.3 are *general* as they catch the semantic assumptions on data constraints in code logic. We also studied Rails (Ruby), Flask (Python), and Hibernate (Java), and they all encapsulate similar sets of APIs for the four database operations. Thus, CFINDER can be migrated to other frameworks or languages with reasonable implementation efforts.

Adding the missing constraints may require extra efforts to clean the data if application data is already erroneous or incompatible. The overhead to perform data cleaning and migration sometimes is not negligible for large tables. However, we consider the effort essential and beneficial because these corrupted data could lead to serious business loss in the future.

Like most issue detection tools, CFINDER still have false positives (§2.4.2) and false negatives (§2.4.3), and there is still space for further improvement. The false negatives could be improved by extending CFINDER with more application-specific code patterns and fine-tuning the static analysis. To avoid the false positives, we would have to rely on developers to manually examine their semantics in code. CFINDER can perform more refinement steps in the static analysis to prune those false positives.

2.7 Acknowledgments

Chapter 2, in full, is a reprint of the material as it appears in the 28th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’23) Haochen Huang, Bingyu Shen, Li Zhong, Yuanyuan Zhou, “Protecting Data Integrity of Web Applications With Database Constraints Inferred From Application Code”. The dissertation author was the primary investigator and author of this paper.

Chapter 3

PYLIVE: On-The-Fly Code Change

3.1 Introduction

Motivated by Python’s popularity and the demand of applying code changes while keeping high availability, this paper presents a new idea that leverages the unique language features of Python to perform dynamic code changes. Specifically, we design and implement a framework, called PYLIVE, that enables dynamically changing Python programs for on-the-fly logging, profiling, and bug-fixing on *production systems* without restarting them.

PYLIVE’s capability to change code during production runs can be used by online services for various purposes including:

- (1) On-the-fly logging for diagnosing production-run errors.** When an online service exhibits some abnormal behaviors, engineers can use PYLIVE to dynamically add logs at certain locations to collect debugging information from production-run. The logs can be enabled only during certain time (e.g. when the load is light) to minimize the performance impact. Production-run information is useful for diagnosing challenging issues (e.g., resource leaking) that are hard to reproduce in a testing environment and only manifest after a long-time (e.g., weeks) running.
- (2) On-the-fly profiling for diagnosing production-run performance issues.** When an online

service has performance issues observed for certain types of requests, engineers can use PYLIVE to dynamically profile a set of functions for a short period of time to collect production-run timing information to troubleshoot the issues. The capability to (1) dynamically start and stop profiling during production runs and (2) only profile a small set of specified functions allows engineers to troubleshoot elusive performance issues without introducing much performance overhead. These performance issues may only emerge in production-run but are hard to reproduce during offline testing, making it necessary to perform in-production profiling.

(3) Urgent dynamic patching (bug-fixing or security patch). PYLIVE allows dynamically applying urgent patches to fix some critical bugs or security issues without stopping and restarting an online service. These bugs and issues can either cause major failures or open up vulnerabilities to attackers and thus needs to be patched as soon as possible.

PYLIVE complements the commonly-used system update practice—rollout deployment [47, 7, 140]. Rollout is not the best choice for dynamic logging and profiling for two reasons. First, rollout still requires a restart of each service instance, which can clear the key program states for diagnosis. These states (e.g., resource leaks) may only be reproduced after a long production run, which is undesirable to be cleared by the rollout. Second, a rollout deployment is heavyweight and overkill for just collecting logging/profiling information. For instance, sometimes only a few servers of a fleet exhibit abnormal behaviors because of their unique memory states. If engineers want to diagnose the issue by rolling out a patch with new logging statements, this patch needs to be batched together with many other patches and will not be applied until the next deployment (wait for a few hours or even a few days). PYLIVE provides a better solution from both perspectives. It requires no restart of service instances so it retains the issue states for logging/profiling. In addition, it enables dynamically adding logging/profiling statements to a running program quickly and also removes the statements flexibly.

PYLIVE is designed based on the unique language features of Python. Python is an interpreted language that supports the meta-object protocol [89, 22] and dynamic typing. The

meta-object protocol enables programs to dynamically modify their own metadata, including function bodies/interfaces and class attributes, while dynamic typing allows changing variable types during running. This makes dynamic code change much easier for Python than for compiled languages (e.g., C/C++) and other interpreted languages that do not support the full meta-object protocol or dynamic typing (e.g., Java):

- For compiled languages, dynamically changing a program requires many complex transformations to its code (e.g., patch functions) and memory layout (e.g., load new code into memory), as shown in previous work [84, 40, 107, 52, 82, 122, 121]. At run time, a compiled program's code is binary code and its memory layout is fixed in different segments. Modifying binary code or memory layout may introduce safety concerns.
- Some other interpreted languages, like Java, do not support the full meta-object protocol or dynamic typing. For example, Java does not support many types of dynamic changes, like adding/deleting methods or changing methods' signatures. Java is also statically-typed, making it hard to change variable types. For such languages, implementing dynamic code change requires modification to the language runtime (e.g., JVM), as in previous work [149, 111, 123]. This introduces portability concerns as different versions of runtime may be used by different systems.

PYLIVE makes two main contributions: (1) PYLIVE realizes safe and portable dynamic code change by leveraging Python's language features. PYLIVE is safe as it requires no low-level transformations of machine code or memory layout. PYLIVE is portable as it relies only on the interfaces provided by standard Python interpreters and thus can be easily adopted by existing Python-based systems. (2) Besides patching, PYLIVE also enables instrumentation-based code changes for dynamic logging and profiling. PYLIVE provides convenient interfaces to flexibly instrument customized code to a selected set of functions at running time. This is useful for collecting diagnostic information in production-run systems without causing significant

performance degradation.

We evaluate PYLIVE with 20 *real-world* cases from seven widely-used Python-based applications (including the popular Django framework and Gunicorn used by Instagram [34]). All these applications have been deployed in various companies to serve millions of customers [68, 13, 5, 23, 26, 27]. PYLIVE successfully helps resolve the cases with on-the-fly logging, profiling and patching with little overhead. Additionally, PYLIVE has helped two widely-used open source e-commerce applications diagnose two new performance issues. We also measure the performance benefit of PYLIVE by comparing it with restarting services to apply code changes. During normal time (no code change), PYLIVE’s overhead is negligible. Upon a code change, PYLIVE causes no downtime and has little performance degradation ($\leq 0.1\%$). In comparison, restarting the applications to apply changes can cause 2-17 seconds downtime and take up to 4.5 minutes to warm up (up to 90% performance downgrade during warmup).

3.2 Background

This section briefly describes the unique language features of Python that enables PYLIVE’s dynamic code change.

Meta-object protocol. Python is designed with a full support of the meta-object protocol [89, 22]. A meta-object is an object that contains a program’s metadata, including types, interfaces, classes and methods, etc. The meta-object protocol provides programming interfaces for programs to manipulate these metadata at runtime. For example, a new method named A can be dynamically added to class C simply with `C.A = D` (D is A’s definition). Similarly, an existing function’s code body can also be changed at runtime with `A.__code__ = D.__code__`. Once it is changed, the new D is called when A is invoked. Other supported changes include changing a function’s interface, adding a new field to a class, changing a variable’s type, etc. All the above changes are supported by the standard Python interpreter [32].

Dynamic typing. Python defines and checks variable types at running time, which makes it easy to dynamically change them. For instance, a Python variable `a` can be changed from a string to a bool. And at running time each time before `a` is used, a type checking is performed. A wrong typed call `compare(a, b)` is detected by the Python interpreter, which will throw a runtime exception. In comparison, for Java, it is impossible to dynamically change a variable's type without changing the underlying JVM. For C/C++, this is also almost impossible as a string is passed by pointer while a bool is passed by value.

Python bytecode. A Python interpreter stores and interprets programs in bytecode, providing an opportunity for dynamically instrumenting code. Compared to machine code, bytecode is much easier to analyze and change with automatic tools. First, it is architecture-independent. Although Python can run on X86-64, ARMv7 and other architecture, it has the same bytecodes for all architectures. So the same tool can be used for Python on all platforms. Second, bytecode also has fewer instructions than machine instructions. Python 3.8 bytecode only has 112 instructions, while X86 alone has 1503 instructions, let alone all various architectures. Third, Python bytecode retains more type information than machine code.

3.3 PYLIVE Framework

PYLIVE is a runtime framework that accepts dynamic change requests from engineers and applies them dynamically into production-run systems without a restart. PYLIVE can be used for on-the-fly logging, profiling and bug-fixing.

In this section, we begin with the design objectives (§3.3.1) and interfaces (§3.3.2) of PYLIVE. We then discuss the three challenges faced by PYLIVE: (1) How to support dynamic changes for function interface, function body and data structure? (§3.3.3) (2) How to identify safe change points to apply a change without causing inconsistency problems? (§3.3.4) (3) How to update programs with multi-threads and multi-processes? (§3.3.5)

3.3.1 Design Objectives

PYLIVE is designed with the following objectives:

(1) *General.* PYLIVE's generality comes from three aspects: (a) it requires no change to the standard Python interpreter. Therefore, engineers need not to download a modified interpreter, which may not be compatible with their systems. (b) PYLIVE is also general on the types of changes supported. It supports not only changes to function body, but also changes to function interfaces (e.g. add one more parameter) and data structures (e.g. add one more field). (c) Since most online services have multiple threads/processes, PYLIVE also provides support for multi-threads and multi-processes.

(2) *Flexible.* PYLIVE is flexible from two perspectives. First, PYLIVE is flexible in terms of *when* to apply a change and *when* to revert a change, all based on engineers' requirements. This can help engineers collect logs for a short amount of time to minimize the performance impact. For example, they can perform on-the-fly logging or profiling only during light-load time. Second, PYLIVE is also flexible with *where* to profile or log. PYLIVE allows engineers to specify which modules or functions to instrument logging/profiling code.

(3) *Consistent and Safe.* Dynamic changes to a running program need to be performed at a carefully selected execution point (aka, a safe point) to avoid inconsistency problems. For instance, changing an `unlock` function to its new implementation after an old `lock` function is already executed may cause inconsistency, leading to incorrect states. Unfortunately, choosing a safe point for general changes has proved to be undecidable [79]. So PYLIVE relies on engineers' knowledge to decide when a change is safe to happen: either when the changed functions are not executing, or a user-specified check function (e.g., specifying a lock is not held) returns true.

(4) *Low Overhead.* PYLIVE is designed to impose as little overhead as possible. At normal time when no change needs to be applied, the PYLIVE thread is sleeping and simply waiting for engineers' inputs. Once engineers instruct it to make code changes, PYLIVE's thread is woken

up to perform the change. Once a change is already applied in this target program’s meta-data, PYLIVE gets to sleep again and is no longer involved in the execution of the target program.

(5) *Little Human effort.* PYLIVE aims to minimize engineers’ efforts in using it. PYLIVE itself is downloaded as a small Python library that can be easily installed. Only two lines of Python code are needed to set up PYLIVE at the initialization of the target program. To insert a dynamic change, PYLIVE only needs engineers to write a small Python snippet to specify what needs to be changed. As shown in our evaluation of 20 real-world cases from seven widely-used Python software systems (cf. Table 3.2), each change specification needs only 7-13 lines of code).

3.3.2 PYLIVE’s Interfaces

To make it easy to use, PYLIVE allows engineers to write the specification for dynamic code change in Python code. To enable dynamic changes for various purposes, PYLIVE supports two change interfaces: `instrument` and `redefine`.

Instrument. The `instrument` interface can instrument code to specified locations in certain functions or modules. It is useful for instrumenting log statements or profiling code to diagnose bugs or performance issues. The interface is:

```
instrument(scope, jointpoint_callback, time).
```

`scope` is a list of function/module names that need to instrument. When only the module name is given, all functions in it are instrumented. `jointpoint_callback` is key-value dictionary of jointpoints (instrument location) and callback code to instrument. PYLIVE supports different granularity of jointpoints: coarse-grained, such as function begin/end, and fine-grained, such as before/after a line and before/after a variable’s definition. This allows engineers to flexibly customize the instrument locations. `time` allows engineers to specify when to perform an

instrumentation and when to revert the instrumentation. `time` can be either a specific time or a function that decides if an instrumentation should be performed. Engineers may want to profile an online service only when it is lightly-loaded and only for a period of time.

Redefine. The `redefine` interface is for code changes that replace the definitions of existing functions and classes (data structures) with new ones. To perform such code changes, engineers use the following Python interface:

```
redefine(preFunc, old_new_map, safepoint).
```

`preFunc` is a user-defined Python function that engineers need to provide to execute before making the specified change. Inside `preFunc`, engineers can import new modules and perform various initialization tasks. `old_new_map` is a key-value dictionary that specifies the changes. Each pair

```
{'old_func/old_class': new_func/new_class}
```

specifies an old function or class needs to be replaced by a new function or class. Engineers also need to provide the new function or class definition. To add a field to a class, just specify the field name and its initialization code with:

```
{'class.new_field': field_init}.
```

`safepoint` defines at what execution point it is safe to apply the specified change. It can be either "FUNC QUIESCENCE" or a user-specified consistency check function (cf. §3.3.4).

3.3.3 Support Dynamic Changes

Change function interface and body. PYLIVE supports changes on both function interfaces and code bodies. Changing function interfaces includes altering the number of parameters, parameter types, and function names. To make these changes, PYLIVE utilizes Python's meta-object protocol and dynamic typing. For parameter number changes, Python functions' parameters are defined in a list (i.e. `co_varnames`), and PYLIVE directly edits the list to add or remove parameters. For parameter type changes, Python uses dynamic typing, and so PYLIVE needs not explicitly change anything. For function name changes, PYLIVE defines a new function and modifies the callers to call the new function.

For function body changes, PYLIVE supports two types of changes: redefine a function body with a new one and instrument the old function body with extra statements (e.g., for logging or profiling). For both types, PYLIVE replaces functions' code object (`__code__`) as a whole, as code objects are immutable and can only be replaced by reference change. For instrument changes, PYLIVE first copies the function's code object, builds an instrumented version by modifying its bytecode [10], and then sets the function's `__code__` to point to the instrumented code object.

PYLIVE may also need to change caller functions when changing the callee functions. For changes that modify callees' interfaces, PYLIVE needs to change all the callers' function body to call the new interface. PYLIVE expects that engineers include all changes to callers in the same patch as normal patching practice. For changes that only modify callees' function bodies, PYLIVE needs not to change the callers. This is because Python function calls are made by function names instead of addresses. Every time a function call happens, Python interpreters translate its name into address by looking up its metadata. Therefore, as long as the function metadata is updated (e.g. modify the `__code__` as discussed before), function calls can always be directed to the newest code objects. Note this differs from dynamic code changes in C/C++—they may need to update callers' function body as the function calls are made by address directly.

Change data structure. Data structure changes include changes to class attributes, object attributes and methods.

Class attributes are data fields defined in classes and shared by all the object instances. PYLIVE changes class attributes by modifying the namespace tables of the target classes. In order to hook class attributes access for profiling or debugging, PYLIVE adds `getter` and `setter` functions for attributes need to be changed. In Python, `getter` and `setter` are automatically called if an attribute is annotated as property.

Object attributes are more difficult to change since they are individually stored in different objects even though they are instantiated from the same class. In order to change an object attribute, it is necessary to go through all objects of the class and change each individually. Previous works typically need to refactor a system ahead of time so they can have Factory objects to keep track of all live objects at runtime [44]. PYLIVE utilizes Python's garbage collector (GC) to track live objects and modify each one when a change is requested. Specifically, PYLIVE calls `gc.get_objects()` to obtain a list of all live objects tracked by GC [14]. As Python uses reference counting to decide objects' liveness, this does not trigger a heap walk but returns a list immediately instead.

Methods are just functions defined in classes and so can be changed in the same way as global functions as described above. Methods' code is only stored in their classes instead of all instantiated objects, and so simply updating the classes' methods is sufficient to apply a change.

3.3.4 Identify Safe Change Point

Changing code at run time is not always safe. For instance, changing a function when it is executing may cause inconsistency problems. Therefore, dynamic code change systems need to carefully choose a safe execution point to apply a change. Unfortunately, choosing a safe point for general changes has proved to be undecidable [79]. As a result, it is necessary to have engineers' knowledge to choose a safe change point. PYLIVE categorizes safe points into different types and

lets engineers select one based on the changes they want to make. Note that choosing the safe update point is only necessary when applying patches. PYLIVE can always apply code changes for logging and profiling as they only add code but do not change the existing code. PYLIVE supports two kinds of safe points:

Quiescence of the changed functions. This requirement means a change is only applied when the changed functions are not under execution. This is also the update point used by many previous dynamic code change systems [40, 41, 40, 143, 150]. It ensures that no function is executed with a mixture of old and new code during changes. PYLIVE provides automatic support for this safe point. To specify it, engineers only need to specify `safepoint='FUNC_QUIESCENCE'`.

PYLIVE supports function quiescence for both changing one function and multiple functions. When changing one function, PYLIVE directly takes advantage of Python meta-object protocol to guarantee the quiescence. In Python, when a function's code is changed, the change only takes effect the next time it is called. When changing multiple functions, PYLIVE checks every thread's stack for any changed function. If any changed function is on a stack, PYLIVE defers the change, retries the checks later and applies the change when no changed function is on any stack.

Consistent state check. When the changed functions modify shared states between them, function quiescence may not be enough for safety. Consider an example shown in Figure 3.1, two functions `lock` and `unlock` need to be changed, and both of them modify the lock state. Applying the change when the program is executing between the calls to `lock` and `unlock` is not safe, even though the functions themselves are not executed. The new `unlock` may be called with an old lock state and the behavior is undefined.

To address this, PYLIVE allows engineers to provide a customized boolean function to decide when it is safe to apply a change. This is also noted as state quiescence in previous work [73]. Engineers can easily write such boolean functions in normal Python code. Figure 3.2


```

30 def file_move_safe(old_file_name, new_file_name):
    ...
58 fd = os.open(new_file_name,...)
59
60 try:
61     locks.lock(fd, ...)
    ...
65     os.write(fd, ...)
    ...
66 finally:
67     locks.unlock(fd)
    ...

```

Unsafe points
for changing
lock() / unlock()

Figure 3.1: An example of unsafe change points for a patch from Django [11]. It is unsafe to change `lock()` or `unlock()` when the program is executing between line 61 and 67, as the change can cause that a new `unlock()` to be called against an old lock, which can lead to undefined behavior.

```

def state_check_func():
    for fd in all_fds():
        if locks.check_lock(fd) != locks.UNLOCK:
            return False
    return True

```

Figure 3.2: An example of state check function for the patch in figure 3.1. It returns true when the lock is not currently held.

shows the state check function for changing `lock` and `unlock`. It checks if no lock is held before applying the change. PYLIVE periodically evaluates it and only applies the change when it returns true.

Note for most code changes, it does not require any customized consistency check. In our evaluation with 20 real-world cases from seven widely-used Python programs, only a few cases require a simple consistency check.

Guidance for engineers. We provide guidance to help engineers identify and specify safe change points for their needs:

First, if the changed functions have no side effects or negligible side effects on execution state, engineers can specify function quiescence as the safe change point. For example, if the changed functions modify no non-local variables, perform no database write and only write a few

logs, it is safe to update them as long as they are not under execution.

Second, if the changed functions have some non-negligible side effects on execution states, engineers need to identify the states that the side effects of the old and new functions will not affect each other. Specifically, the variables V_{old} defined and propagated from old functions f_{old} will not be used by new functions f_{new} , and vice versa. To ensure this, the target consistent states are either no variable in V_{old} is defined or all of them are dead. An example of this is shown in Figure 3.2 that no lock is held at the point of change. Such states may not exist or may not be easy to express in state check functions, and in such cases it may be better to perform a restart than to use PYLIVE.

3.3.5 Support for Multi-threads and Multi-processes

Multi-threads. A server program may have multiple threads to serve different user requests. Different threads have different program counters while sharing the same code and global variables. Therefore, it is not straightforward to apply a change at a given safe change point for multiple threads.

To change multiple threads correctly, PYLIVE applies a given change synchronously. The synchronous change is ensured by Python global interpreter lock (GIL). At any execution point, only one thread can hold GIL and so can get executed [15]. Therefore, when PYLIVE is actively applying a dynamic change, all other threads to be changed are blocked. When applying changes, PYLIVE also explicitly holds GIL lock to make sure no other threads can preempt it [17].

Based on the type of safe point, PYLIVE applies changes differently. If the safe point is function quiescence, PYLIVE either immediately applies the change when only one function needs to change or check program stacks to make sure no target function on stacks when multiple functions need to change. Applying one function change is simpler because Python's meta-object protocol ensures the change to not take effect during its execution. If the safe point is a consistent state check, PYLIVE first executes the check function provided by engineers. If the consistent

check succeeds, PYLIVE then applies the change. If it fails, PYLIVE sets a timer t and goes to sleep to let other threads execute. The timer will wake up PYLIVE later to perform a state check again. The timer t is configurable by engineers. After several attempts, if it still fails, PYLIVE will give up and report an error to engineers.

Multi-processes. Online services may use multiple processes, and dynamic code change needs to be applied to all of them. Different Python processes reside in different address spaces and share code through copy-on-write. When code is changed in a process, a copy-on-write happens and other processes will continue to use the old code. As such, dynamic code change needs to be performed explicitly in all processes. PYLIVE adopts a controller-stub architecture to communicate changes to all processes. A stub is a change thread residing in a target process. PYLIVE starts one stub thread for each target process at its starting time. A stub thread listens to a controller for patches and applies the received patches at a safe change point. A PYLIVE controller is a standalone process that accepts engineers' change input and sends the specified code change to the stub thread in each process.

3.4 Use Cases

PYLIVE enables three types of use cases that require a running system to be dynamically changed.

3.4.1 On-the-fly Logging for diagnosis

Systems may exhibit abnormal behavior during running. To collect run time info for diagnosis, engineers may want to add new log messages dynamically without restarting services.

An example of this is the diagnosis of a bug [28] from the Shuup [24] e-commerce system. This bug is related to its shopping cart: when some users click "add to cart", the product is not

```

# callbacks to instrument
# logging right/left-hand variables in each line
def call_b(_righthands):
    logging.info(_righthands)
def call_a(_lefthands):
    logging.info(_lefthands)

# instrument code to every line in two functions
instrument(scope=['...add_product',
                 '..._find_product_line_data'],
          jointpoint_callback={line_before: call_b,
                              line_after: call_a},
          time='24:00-2:00')

```

Figure 3.3: PYLIVE’s dynamic logging spec for an urgent, real world bug in Shuup e-commerce system [28]. This spec tells PYLIVE to dynamically instrument code to log some variable values in two functions `add_product` and `_find_product_line_data` for a period of time. `line_before` and `line_after` are two jointpoints PYLIVE provides to instrument code before and after each line in functions.

added to the cart. This prevents users from purchasing products and causes direct revenue loss to businesses. Since the bug has no error logs, it is quite challenging for engineers to diagnose it off-line.

Figure 3.3 shows how engineers can use PYLIVE to add log messages to diagnose the issue. Engineers direct PYLIVE to add line-by-line logs in two functions `add_product` and `_find_product_line_data`. Engineers also specify to only collect logs during light-load time (24:00-2:00).

3.4.2 On-the-fly Profiling

Performance issues often occur in production as systems have more and more features and scale up to a larger size. When such an issue emerges, engineers may want to enable profiling to certain parts of a system during production run.

An example [21] of such issues is from the Oscar e-commerce system. This issue happens when there are a lot of product categories in Oscar. The issue causes a performance downgrade in many pages displayed to customers in Oscar, preventing customers from buying products.

```

# profiling code to instrument
def call_b(start):
    start = time.time()
def call_a(start):
    logging.info(time.time()-start)

# instrument code to all functions of two classes
instrument(scope=['...AbstractCatagory.*',
                 '...CatalogueView.*'],
          jointpoint_callback={func_before: call_b,
                              func_end: call_a},
          time='24:00-2:00')

```

Figure 3.4: On-the-fly profiling using PYLIVE to diagnose a critical performance issue occurred in Oscar e-commerce system [21]. This example requires PYLIVE to instrument code to profile the execution of every method in two classes `AbstractCatagory` and `CatagolueView` for a period of `time`.

Figure 3.4 shows how to use PYLIVE to dynamically instrument code to profile the system. Engineers instruct PYLIVE to instrument customized profiling code into the methods in two classes, `AbstractCatagory` and `CatalogueView`, that are speculated to be related to the issue.

3.4.3 Dynamic Patching

Online services frequently have urgent bugs (e.g., security bugs) that need to be patched as quickly as possible to minimize damages since they may cause information leakage/system compromise and prevent customers using online services.

An example [1] of such patches is from Django. It fixes a severe cross-site scripting (XSS) [8] issue, CVE-2019-12308 [9]. The issue is scored as “6.1” since it can expose malicious URLs as clickable links to victim users and direct them to vulnerable sites. Django developers quickly post a security release [12] to fix the vulnerability and encourage all online services that use Django to apply it as soon as possible.

Figure 3.5 shows part of the patch and the change spec that engineers need to provide for PYLIVE to dynamically apply it. This patch is non-trivial to be dynamically applied, as it changes both function interfaces and data structures. It adds a new parameter `validator_class` to the

```

# patch: add a parameter validator_class
# add an object attribute validator
class AdminURLFieldWidget(...):
    def __init__(self, attrs=None,
                 validator_class=URLValidator):
        self.validator = validator_class()
        ...

# change specs.
def preupdate_call():
    from django.core.validators import URLValidator

    redefine(
        preupdate = preupdate_call,
        old_new_map={
            '...AdminURLFieldWidget.__init__': __init__},
        safepoint='FUNC QUIESCENCE')

```

Figure 3.5: A real world security patch to Django [1] and PYLIVE’s dynamic change spec for it. This patch adds a parameter to function `__init__` and adds an object attribute `validator` to class `AdminURLFieldWidget`. Other part of the patch is omitted due to space limit. The change spec indicates: `preupdate` — `import URLValidator` before the change; `old_new_map` — replace `AdminURLFieldWidget.__init__` with a new one; `safepoint` — apply the change when the changed function is quiescent.

`__init__` function and adds a new attribute `self.validator` to `AdminURLFieldWidget`. The change spec calls the `redefine` interface with three arguments: `preupdate` specifies that PYLIVE needs to import a new class `URLValidator` before applying the change; `old_new_map` indicates that the original `__init__` will be replaced with the new code. `safepoint='FUNC QUIESCENCE'` tells PYLIVE to apply the change when the changed functions are quiescent. This requirement is safe enough in the case as there is no inter-dependency between the changed functions.

3.5 Evaluation

3.5.1 Methodology

We evaluate PYLIVE with 20 cases from seven Python-based real-world applications, as shown in Table 3.1. These applications are deployed in many companies, serving millions of

Table 3.1: 20 real-world cases evaluated in our experiments. They are from seven widely used Python-based server applications that have powered many commercial e-commerce and ad-based web services including Instagram, serving millions of customers.

Applications	Category	Logging	Profiling	Patching
Django [33]	Web framework	1	0	2
Gunicorn [16]	Web server	0	0	1
Oscar [6]	E-commerce	1	2	1
Odoo [25]	E-commerce	1	1	2
Shuup [24]	E-commerce	1	0	1
Pretix [27]	E-commerce	1	0	1
Saleor [92]	E-commerce	1	1	2
Total		6	4	10

customers [68, 13, 5, 23]. Django is a popular web framework that powered over 94,319 websites, of which many are for e-commerce [142]. Gunicorn is a production web server used by many big companies for their main services, such as Instagram [34]. All the online services need to be almost non-stop since any downtime can result in revenue loss.

To evaluate PYLIVE’s benefit, we compare PYLIVE with a typical restart approach: modify code for logging/profiling/patching offline, stop the services and restart the services *immediately*. To precisely measure the restart impact, we only restart the Python part of a service, which does not restart other parts (e.g., database) to avoid the impact of warming up their cache. For profiling, we also compare PYLIVE with cProfile [72], which is Python’s official profiling tool for collecting comprehensive profiling information in test environments. Note PYLIVE is not a substitute for cProfile as it collects less information than cProfile. However, as we will present in the results, some cases only need little dynamic information to diagnose. We conduct this comparison to study the benefit that PYLIVE can bring for such cases.

Each application is set up on a machine with a 2.30GHz CPU (6 core), 16GB Memory and 256GB SSD. Each application runs with 2 processes and 4 threads/processes. Each application is initialized with ~2000 web pages. To mimic real-world workloads, JMeter [4] is used to generate random web page accesses. The JMeter client is started with 8 threads and can generate up to

15K requests/second.

We use throughput as the performance metric and normalize it to the max throughput of normal service run (41-752 requests/second). All the experiments are conducted within a LAN, which ensures that the network is not the bottleneck.

3.5.2 Overall Performance Results

Overall, PYLIVE avoids 2-17 seconds downtime and avoid up to 4.5 minutes warmup time, during which the performance downgrade can be 55%-90%. PYLIVE causes negligible (<0.1%) overhead during normal run as well as applying changes. PYLIVE causes 0.1%-1.4% overhead during profiling. Compared with cProfile, PYLIVE's selective instrumentation avoids 10.5%-33.6% overhead.

Figure 3.6,3.7 show the results of eight representative cases. Two newly identified performance issues and the other twelve existing real-world cases have similar results and due to space limit are put online [3].

For **logging cases**, PYLIVE's benefits mainly come from avoiding the time to restart and warm up. The service restart is relatively fast (2-17 seconds), but the warmup takes much longer time. Our experiments set up applications with only ~2000 web pages, but the warmup still takes 2.3-3 minutes.

For **profiling cases**, PYLIVE makes the performance impact caused by profiling affordable in production-run systems. The benefit comes from two aspects. First, PYLIVE allows engineers to perform customized profiling, so they need not profile applications in a whole as with cProfile. The customized profiling is not a substitute for comprehensive profiling with cProfile because it collects less information. However, it's sufficient to diagnose many cases that only needs limited timing information, as shown later in our case studies (cf. §3.5.3). Second, PYLIVE avoids restart and warmup time (up to 4.5 minutes), which is needed by cProfile. With PYLIVE, the performance downgrade during profiling is 0.1%-1.4%. While with cProfile, the performance

downgrade can be 11%-39%.

For **patching cases**, PYLIVE can apply them dynamically with almost no performance downgrades. This benefits urgent security patches, for which waiting for the next rollout can be dangerous. Our evaluation includes 5 security patches and PYLIVE successfully applied them on-the-fly.

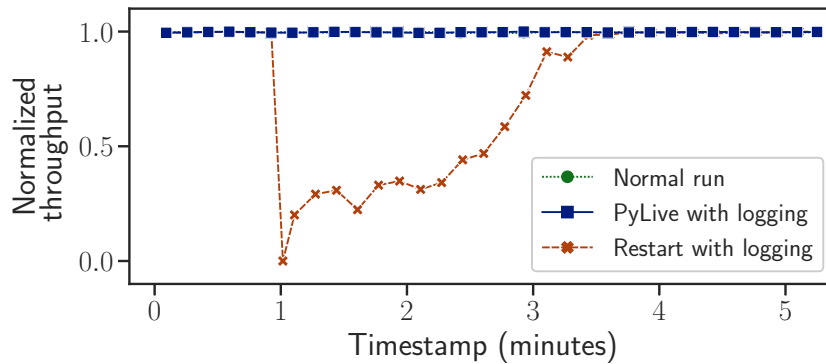
3.5.3 Case Studies

This section dives into the details of eight representative cases. The remaining twelve cases evaluated are similar and due to space limit we put them online [3].

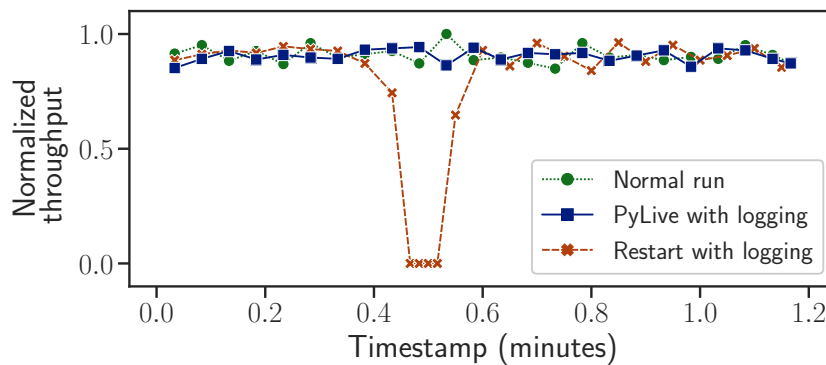
Case 1: Diagnose a purchase bug in Shuup [28]. This case is about diagnosing a bug related to the shopping carts of Shuup [24], a widely-deployed e-commerce website. As mentioned in §3.4.1, the bug causes an error in production and prevents customers from adding new items to shopping carts. To help diagnose it, PYLIVE dynamically instruments logging statements on the running application. Figure 3.6a shows that PYLIVE avoids 3 seconds downtime and 2.3 minutes warmup time. It imposes only ; 0.1% performance overhead.

Case 2: Diagnose a payment bug in Odoo [114]. Odoo [25] is an e-commerce website and this bug prevents customers from paying an order. It is an “urgent” bug as it results in business loss. Odoo engineers diagnosed it by adding two logging statements and restarting the service. With PYLIVE, the logging statement can be added on-the-fly with 11 LOC. As shown in Figure 3.6b, PYLIVE avoids 4 seconds service downtime with ; 0.1% overhead. Differing from other applications, Odoo does not have much cache and so requires little warmup time.

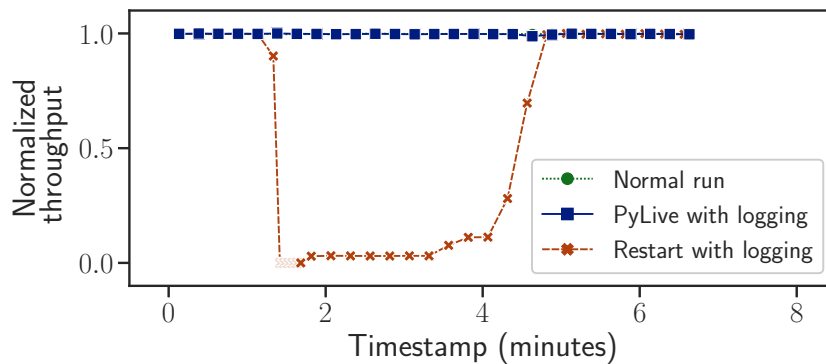
Case 3: Diagnose a purchase bug in Pretix [20]. Pretix [27] is a ticket-booking website that allows event organizers to sell event tickets online. In this case, when customers request a PayPal refund, it fails silently with no error messages. PYLIVE can dynamically instrument logging code to diagnose the reason. Figure 3.6c shows that PYLIVE successfully avoids 17 seconds downtime and 3 minutes warmup (by restarting Pretix) with ; 0.1% performance overhead.



(a) **Shuup: on-the-fly logging to diagnose a payment bug [28].** PYLIVE causes $\approx 0.1\%$ overhead only when adding logs. In comparison, restarting causes 3 seconds of downtime and needs 2.3 minutes to warmup.

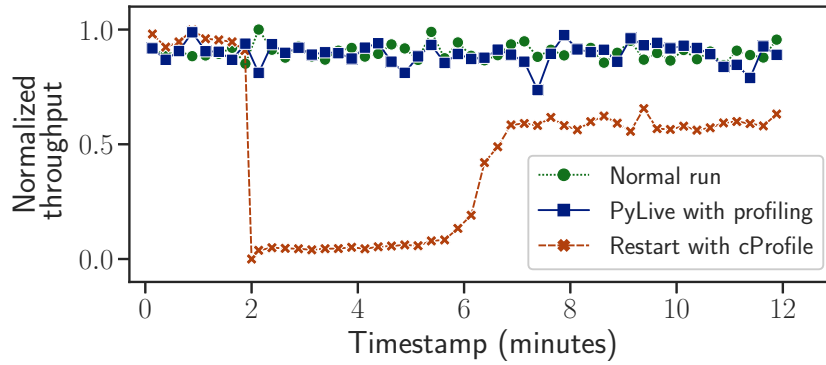


(b) **Odoo: on-the-fly logging to diagnose a shopping-cart bug [114].** PYLIVE causes $\approx 0.1\%$ overhead only when adding logs. In comparison, restarting causes 4 seconds of downtime. Odoo does not have much cache so has a short warmup time.

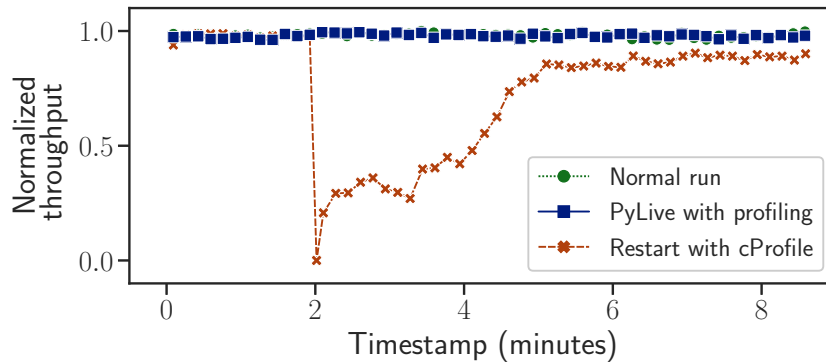


(c) **Pretix: on-the-fly logging to diagnose a payment bug [20].** PYLIVE causes $\approx 0.1\%$ overhead only when adding logs. In comparison, restarting causes 17 seconds of downtime and needs 3 minutes to warmup.

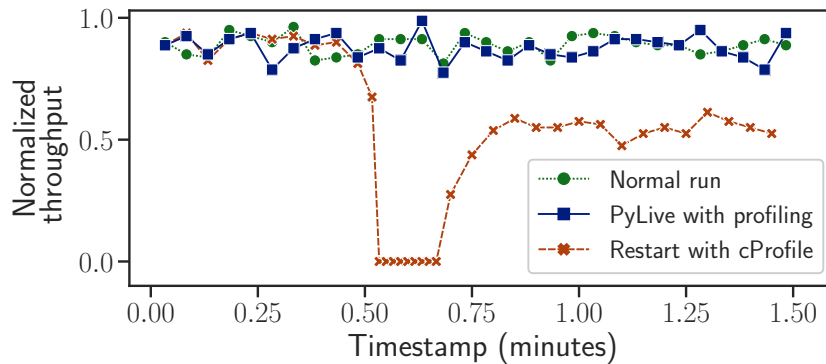
Figure 3.6: Throughput comparison of **three on-the-fly logging cases and three on-the-fly profiling cases** with PYLIVE in comparison with today’s practices—stop and restart with logging added and profiling enabled.



(d) **Saleor: on-the-fly profiling a long-loaded web page [135].** PYLIVE causes 1.4% overhead only during profiling. In comparison, restarting causes 3 seconds of downtime and needs 4.5 minutes to warmup. Using cProfile causes 35% overhead.



(e) **Oscar: on-the-fly profiling a long-loaded web page [21].** PYLIVE causes 0.5% overhead only during profiling. In comparison, restarting causes 2 seconds of downtime and needs 3 minutes to warmup. Using cProfile causes 11% overhead.



(f) **Odoo: on-the-fly profiling a long-loaded web page [115].** PYLIVE causes 0.1% overhead only during profiling. In comparison, restarting causes 9 seconds of downtime. Using cProfile to profile causes 38.5% overhead.

Figure 3.6: Throughput comparison of **three on-the-fly logging cases and three on-the-fly profiling cases** with PYLIVE in comparison with today’s practices—stop and restart with logging added and profiling enabled. (cont.)

Case 4: Profile a main web page in Saleor [135]. This case is about diagnosing a slowly-loaded web page. This case is difficult to reproduce in testing as it only emerges when the product category grows to large. Currently, engineers use cProfile to profile the whole application [135]. Enabling cProfile needs an application restart, causing downtime and warmup time as shown in Figure 3.6d. Also, cProfile profiles every function, so after warmup it still imposes 35% performance downgrade.

PYLIVE can benefit the diagnosis in two ways. First, it can dynamically instrument profiling code into a running application. Figure 3.6d shows this can avoid 3 seconds downtime and 4.5 minutes warmup of Saleor services. Second, it can be customized to only profile the relevant functions suspected by engineers and thus reduces profiling overhead to only 1.4%.

Case 5: Profile a slowly-loaded web page in Oscar [21]. This case is about diagnosing a slowly-loaded product-listing page. It happens when the number of products grows to large. PYLIVE enables dynamic profiling to Oscar with 9 LOC to specify the change. As shown in Figure 3.6e, PYLIVE causes only 0.5% performance overhead during profiling and nearly no overhead during normal run. In contrast, cProfile causes as much as 11% performance downgrades as well as 2 seconds of downtime and 3 minutes warmup time.

Case 6: Profile a slow action in Odoo [115]. This case is about diagnosing a slow receipt-validating action. It is hard to reproduce in testing as it only emerges when the database contains a large number of products and orders. PYLIVE enables dynamic profiling with 9 LOC. Figure 3.6f shows PYLIVE's performance benefit. PYLIVE causes only 0.1% performance overhead during profiling and nearly no overhead during normal run. In contrast, cProfile causes as much as 38.5% performance downgrades as well as 9 seconds of service downtime.

Case 7: Patch CVE-2019-12308 security vulnerability in Django [1]. This patch fixes a severe XSS security issue CVE-2019-12308 [9]. As we discussed in §3.4.3, it may lead users to click into malicious websites and can possibly affect many users. This patch is non-trivial to be dynamically applied, as it involves adding a parameter to a method interface and adding a

Table 3.2: Lines of code (LOC) of change specification for PYLIVE. For patches, this only count extra code for PYLIVE.

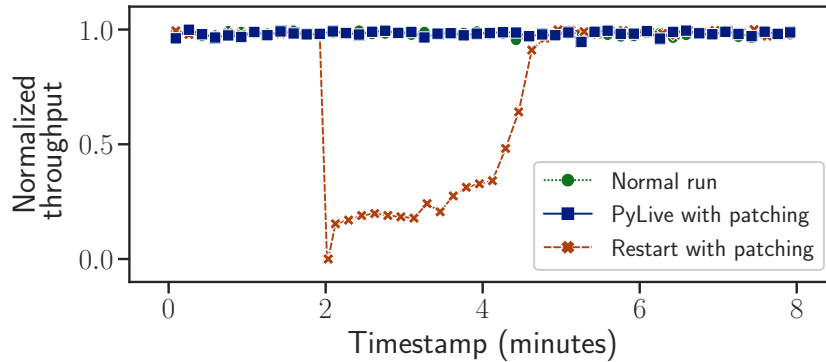
Use Case	Software	LOC	Use Case	Software	LOC
Case1	Shuup	7	Case5	Oscar	9
Case2	Odoo	11	Case6	Odoo	9
Case3	Pretix	9	Case7	Django	8
Case4	Saleor	9	Case8	Gunicorn	13

new class attribute [1]. With PYLIVE, this patch is allowed to be applied safely with 8 additional LOC. Figure 3.7a shows the performance benefit of PYLIVE and PYLIVE avoids 2 seconds of downtime and 2.8 minutes warmup time. PYLIVE dynamically applies the patch with ; 0.1% overhead.

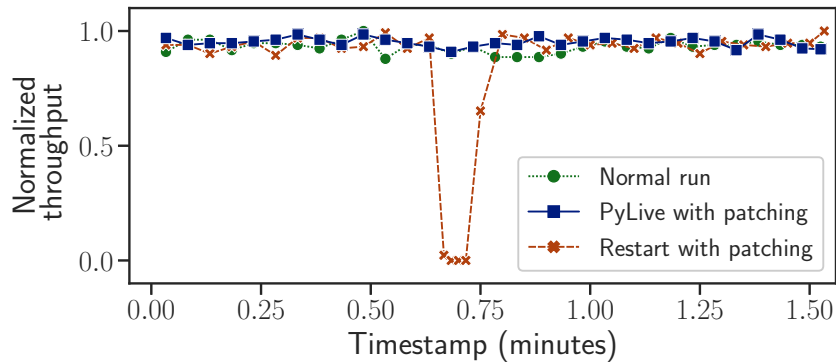
Case 8: Patch CVE-2018-1000164 in Gunicorn [78]. This patch fixes a HTTP Response Splitting Vulnerability [112]. It has a severity score of “**7.5 High**” in the CVE system [113]. It can be exploited by various attacks, such as Cross-site Scripting (XSS), Cross-User Defacement, Hijacking [112]. The patch requires a modification to a function body. It can be dynamically applied with PYLIVE with only 13 additional LOC. As shown in Figure 3.7b, PYLIVE avoids 4 seconds of downtime, when a non-cached service runs on Gunicorn. PYLIVE introduces ; 0.1% overhead while applying the patch.

3.5.4 Human Effort

PYLIVE requires only a little human effort to adopt it in real-world applications. To enable PYLIVE in a Python-based application, it only needs to add two lines of code in the application’s initialization stage. To apply dynamic change for different purposes, PYLIVE allows engineers to write Python code to specify the intended changes. Table 3.2 shows the lines of code (LOC) to specify the changes in the eight representative cases. For all cases, it requires only 7-13 lines of code to specify each change.



(a) **Django: urgent security patching for CVE-2019-12308 [1].** Compared with restarting, PYLIVE avoids 2 seconds of downtime and 2.8 minutes warmup time, with 0.1% performance overhead during patching.



(b) **Gunicorn: security patching for CVE-2018-1000164 [113].** Compared with restarting, PYLIVE avoids 4 seconds of service downtime, with negligible overhead during patching. Gunicorn’s workload is Odoo, which has little cache, so it takes a short time to warmup.

Figure 3.7: Throughput comparison of two representative patching cases with PYLIVE and restarting services.

3.6 Limitations and Discussion

There are many kinds of code changes that PYLIVE cannot apply. First, PYLIVE cannot apply changes to long-running functions because dynamic changes only take effect next time when the functions are called. Fortunately, online services are usually request based and the major part is the request handling functions, which finish running in a short amount of time. Second, PYLIVE cannot apply patches that assume an initial program state. Patches for memory-leak bugs

may need to reinitialize the program state to free the leaked memory, which needs a restart of the target program. Third, PYLIVE is not suitable for applying major changes to a target program. Such changes include adding new features, updating library versions, and refactoring the program structure. These changes may involve major changes to program states and code logic of many functions. Therefore, it is hard for engineers to write code to initialize the states and to specify a safe update point that considers all the dependencies between the changed functions.

PYLIVE relies on engineers to specify the safe update points for dynamic patching. PYLIVE targets on simple bug-fixing and security patches that only update a few functions and data structures. For these patches, the safe update points can be specified as when the targeted functions are not executing or when a customized state check passes (e.g. a lock is not held as in Figure 3.1). However, for more complex patches that change many interdependent functions and data structures, the safe update point may not be easy to specify. For such cases, it is safer to restart the target program than to use PYLIVE. Note the safe update point is only necessary for applying patches but not for logging or profiling. Code changes for profiling and logging can always be safely applied as they only add code but do not change the existing code.

PYLIVE cannot prevent errors introduced by buggy patches. PYLIVE expects that engineers thoroughly test their patches in a testing environment before dynamically applying them to production-run systems. For logging and profiling cases, PYLIVE wraps the instrumented code in try-catch blocks so that buggy logging or profiling code does not affect the normal program execution.

PYLIVE has two security implications. First, in terms of the type of code changes that can be made dynamically, PYLIVE does not expand the attack surface of Python's own meta-object protocol. PYLIVE does not modify the Python interpreter to enable more types of code changes but just provides convenient interfaces purely based on Python's meta-object protocol. Second, the introduction of a change controller (cf. §3.3.5) expands the attack surface from one single process to two processes. The change controller is an additional process that commands a target

program process to apply a change dynamically. Therefore, it would be dangerous if attackers gain access to the change controller. It can be mitigated by setting the change controller's permission to make it only executable by a privileged user. We also plan to implement PYLIVE's own access control for the change controller in future work.

PYLIVE's design and implementation are generally applicable to Python variants and other interpreted languages as long as they support three language features:

- Meta-object protocol—PYLIVE uses this to modify a program's code at running time (cf. §3.2);
- Dynamic typing—PYLIVE relies on this to modify variable types at running time (cf. §3.2);
- Interpreter interfaces to freeze non-current threads—PYLIVE uses them to pause the execution of any other threads to safely apply changes (cf. §3.3.5).

All three features are supported by popular python variants including Pypy [29] and Pyston [2], and so PYLIVE can be easily ported to them. PYLIVE can also potentially be ported to two other popular interpreted languages: JavaScript [19] and Ruby [30]. The first two features are directly supported by JavaScript and Ruby. The third feature can also be implemented in JavaScript and Ruby in different ways. JavaScript uses a single-threaded event loop model—at any time only one event handler is running and it cannot be preempted before its completion. Therefore, when PYLIVE is running in JavaScript, any other thread is ensured not running at the same time. Ruby's official interpreter YARV [31] has a similar GIL lock as Python's GIL, which allows PYLIVE to hold GIL in Ruby to prevent preemption as in Python (cf. §3.3.5).

3.7 Related Work

Dynamic Code Change for C/C++ and Java. Many works have been done for dynamic changing C/C++-based operating systems [143, 44, 43, 51, 99, 41, 74] and applications [85, 40,

108, 82, 52, 98, 148, 133]. While simple dynamic change (e.g. patch only function bodies) to OS kernels has been used in production, more general change to applications has not been widely adopted. General dynamic change usually requires many unsafe transformations to target programs including modifying both machine code and memory layout. This may introduce safety concerns in production. Contrarily, PYLIVE realizes general changes by utilizing Python’s standard language features—meta-object protocol and dynamic typing, making it safer to be adopted in production.

Works on dynamic changing Java either need to modify JVM [149, 111] or rely on some unsafe operations of JVM [123], introducing portability and safety concerns in production. When running a Java program, JVM maintains many metadata such as method signatures and class attributes as internal data but provides no safe operations to modify them. However, it is necessary to modify these metadata in order to support general dynamic change. In comparison, PYLIVE makes use of Python’s meta-object protocol to safely modify related metadata when dynamically changing Python programs. This imposes no modification to a standard Python interpreter and so can be easily adopted in existing production systems.

Dynamic Code Change for Python. Pymoult [101] made a preliminary exploration on the feasibility of dynamically changing Python programs. As a preliminary proposal, it has no experimental result. More importantly, Pymoult relies on a special Python interpreter, Pypy, which is not fully compatible with the standard Python interpreter (i.e. CPython [32]). In order to use Pymoult, engineers need to port their systems to Pypy interpreter, which requires considerable human efforts. Contrarily, PYLIVE is based on the standard Python interpreter and so can be easily adopted in the field.

PyReload [152] is a dynamic code change tool based on the standard Python interpreter. However, it has two key limitations that prevent it to be practical. First, PyReload needs engineers to refactor a target program into modules, which requires huge human efforts. Second, PyReload

only supports single-threaded programs. Considering that servers usually have multiple threads, PyReload is not suitable for server systems. In contrast, PYLIVE requires no refactor to the target program and supports updating multi-thread server programs.

Language level support for dynamic code change. Language level support for dynamic code change is not new. Besides Python, many other dynamically-typed programming languages, such as Common Lisp [147], Smalltalk [75], JavaScript [19] and Ruby [30], have provided support for meta-object protocol. Meta-object protocol can be directly used to update a single function/class; however, there are still many challenges in using meta-object protocol to practically update server programs, which usually needs to update multiple functions/classes and threads/processes. Very few works have been done on these challenges. Rivet [104] proposes interesting ideas to leverage JavaScript’s reflection capabilities to debug single-threaded browser-side programs. But the ideas cannot be directly borrowed to update server programs, which usually have multiple threads/processes.

Focusing on Python, PYLIVE addresses three challenges of leveraging meta-object protocol to update server programs. First, to make it easy to update multiple functions and classes, PYLIVE provides two APIs: `Redefine` and `Instrument` (§3.3.2) and adopts the meta-object protocol and bytecode instrumentation to implement them (§3.3.3). Second, to make it safe to update multiple functions and classes, PYLIVE borrows ideas from previous works [40, 41, 73] and provides two different safe points (§3.3.4). Third, to support updating programs with multiple-threads and multiple-processes, PYLIVE proposes new synchronous update mechanisms based on Python GIL and a controller-stub architecture (§3.3.5).

Aspect-oriented programming. PYLIVE’s `Instrument` interface is a type of aspect-oriented programming (AOP) [91]. AOP is a programming paradigm to break down independent program logic into different modules. A common usage of AOP is to allow developers to write a function’s main logic and its logging code separately. The AOP framework then “weaves” the code together

at compile time or load time. Several works also aim to enable dynamic weaving at run time for AOP [125, 151, 124]. PYLIVE's `Instrument` interface is an AOP support for Python and is inspired by previous work on AOP interfaces for Java [90, 100]. However, AOP's main target is to insert additional code without modifying the existing code. PYLIVE also provides a `Redefine` interface to modify the existing code of a running program, which is challenging especially when the target program has multiple threads/processes. To realize `Redefine`, PYLIVE further considered the challenge of supporting safe update points and multiple-threads/processes programs.

Dynamic Instrumentation. PYLIVE's `Instrument` interface is related to previous works on dynamic binary instrumentation (DBI), including `Pin` [97], `Valgrind` [110], and `DynamoRIO` [48]. DBI enables modifying a running binary program at the machine instruction level and is usually used for logging and profiling a compiled program. However, DBI cannot be directly adopted for profiling a Python program in production. When DBI is used for a Python program, DBI is instrumenting the Python interpreter instead of the Python program. This can cause two folds of problems. First, the logging and profiling results are verbose and hard to understand by Python developers as they are mostly about the execution of the Python interpreter but not about the Python program. Second, this can introduce an unacceptable performance downgrade to the Python program as interpreting one line of Python code may need to run multiple lines of interpreter code. PYLIVE addresses these problems by instrumenting code at the granularity of Python bytecode. Therefore, the logging and profiling results can be directly mapped back to the Python program and so are easy to understand by Python developers. In addition, much less code is instrumented and so much less performance downgrade.

PYLIVE complements `DTrace` [50] on dynamic instrumentation. `DTrace` is a dynamic instrumentation framework for production systems. To enable `DTrace` for Python, it needs to embed “markers” in Python interpreters. This introduces additional compatibility requirements for

Python interpreters to use DTrace. As noted by the official Python document [18], “DTrace scripts can stop working or work incorrectly without warning when changing CPython versions.” PYLIVE takes a complementary approach to instrument Python application code without modifying Python interpreters, avoiding the compatibility concerns.

Rollout Update. An alternative way to avoid whole system downtime is rollout update [47, 7, 140]. In rollout update, a cluster of servers are restarted one by one or batches by batches so that during an update there are still servers alive to serve users’ requests. However, for just collecting logging/profiling information, rolling out patches to a whole cluster at the next deployment is heavyweight and an overkill. It would be handy to quickly apply a simple patch that temporarily logs extra information or collects extra metrics to some servers on-the-fly. Furthermore, rollout update is less effective for collecting diagnostic or profiling information for certain types of issues because rollout update still requires restarting every service instance. As a result, errors that appear only after a long running time, such as resource leaks and concurrency issues, may not reappear quickly after restarting to provide diagnostic information [155]. Finally, rollout update can still result in a subset of servers restarting and warming up before providing service at their full capacity. This means during the rollout update, the entire system will suffer from certain levels of throughput degradation.

3.8 Acknowledgments

Chapter 3, in full, is a reprint of the material as it appears in the 2021 USENIX Annual Technical Conference (ATC’21) Haochen Huang, Chengcheng Xiang (co-first authors), Li Zhong, Yuanyuan Zhou, “PYLIVE: On-the-Fly Code Change for Python-based Online Services”. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Conclusion

This dissertation presents two tools for developers to develop and maintain dependable Python-based database-backed web applications. This dissertation first presents CFINDER to protect the data integrity by automatically inferring the missing database constraints from application source code, leveraging the observation that many source code patterns imply certain data integrity constraints. This dissertation then presents PYLIVE to support dynamic code change without restarting the service by leveraging the unique language features of Python. The example use cases include dynamic logging, profiling, and bug-fixing.

CFINDER focused on the problem of missing database constraints in web applications with resulting data integrity issues and the feasibility of extracting the missing database constraints from the application code. Specifically, we first conducted an empirical study on *missing* constraints in five popular web applications. Then we designed and implemented a tool that identified 210 previously unknown missing constraints with reasonable accuracy from eight widely-deployed web applications, including one commercial company with millions of users. We have reported 92 of them to the developers of these applications, and so far 75 of them are confirmed.

PYLIVE leverages Python’s unique language features, meta-object protocol, and dynamic typing, to support dynamic code change for on-the-fly logging, profiling, and patching in

production-run systems. PYLIVE only relies on standard Python interpreters and can be easily adopted by existing systems. We evaluated PYLIVE with seven widely-deployed Python-based systems for online services. PYLIVE successfully helped resolve 20 existing real-world cases from these systems with dynamic logging, profiling, and patching. PYLIVE also helped two of the systems diagnose two *new* performance issues. In comparison to restart, PYLIVE avoids service downtime and warmup time. PYLIVE imposes no overhead during the normal run and negligible overhead during the code change. For profiling, PYLIVE adds only 0.1%-1.4% overhead.

4.1 Lessons Learned

Finally, I will share some lessons that I learned, which may be helpful for future research and guide future directions.

First, today's web applications become increasingly complex and it's challenging for developers to understand their dependencies. The application could have multiple tiers of components such as the front end, the backend servers, the database, the cache, async workers, etc. And the components may be written in different programming languages and maintained by different teams. As the industry demands more engineers to build various applications, many of them do not have solid backgrounds and training to understand them all. Additionally, to err is human, even experienced developers can easily forget and neglect the dependencies between components due to deadline pressure. As a result, developers make many wrong assumptions about other components, leading to various mistakes in their web application code, and causing severe reliability, availability, and security issues. Neglecting the database constraints in the database layer presented in this dissertation is one example. Some other examples include: (1) inconsistent or missing checks (for both integrity and security) widely exist among multiple tiers, such as the front-end, back-end, and data layer. (2) changes to one component could lead to the failure of another dependent component if neglected and not updated together. More work can be

done to understand the dependencies between the various components, take advantage of each tier to provide the defense in depth, and find better ways to control and manage the dependencies.

Second, more useful semantic information could potentially be inferred from the application source code. Previous work has mainly focused on detecting syntax errors from the source code. However, the code also contains certain patterns that can infer the underlying semantics because the developers make assumptions about the semantics when they write the code. The extracted semantics can be used in a variety of ways to build more reliable applications, such as checking the consistency between components or comparing with developers' specifications. For example, the application code has checks on access control, but they are spread in multiple code paths. Future work can extract these security policies, check their consistency across modules and components, and impose and verify them in a central place.

Third, more work can be done on tooling support for diagnostics and stop-the-bleeding operations in the production environment for web applications. To minimize the time to resolve an incident, it would be very helpful for developers to fix it in production while keeping all the in-memory data. As it's always challenging or even impossible to reproduce the issues in an offline environment. While at the same time, the challenge is that we still want to make sure we don't harm the production environment and make things worse. PYLIVE is a first step in providing the ability to change the running code while keeping the in-memory data without restarting the servers, such as adding more logs. We could potentially support or automate more operations to stop the bleeding, such as adding branches to bypass some unexpected inputs that trigger the errors.

Bibliography

- [1] [2.2.x] Fixed CVE-2019-12308 – Made AdminURLFieldWidget validate URL before rendering clickable link. <https://github.com/django/django/commit/afddabf8428ddc89a332f7a78d0d21eaf2b5a673>.
- [2] A faster and highly-compatible implementation of the Python programming language. <https://github.com/pyston/pyston>.
- [3] Anonymous repo for Pylive evaluation result. <https://github.com/pyupdate/evaluation>.
- [4] Apache JMeter - Apache JMeter. <https://jmeter.apache.org/>.
- [5] Butterfly Network Case Study – Saleor Commerce. <https://saleor.io/case-study/butterfly-network/>.
- [6] Cases / Oscar - Domain-driven e-commerce for Django. <http://oscarcommerce.com/cases.html>.
- [7] Continuous Deployment at Instagram. <https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1>.
- [8] Cross Site Scripting (XSS) Software Attack — OWASP Foundation. <https://owasp.org/www-community/attacks/xss/>.
- [9] CVE-2019-12308 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2019-12308>.
- [10] Disassembler for Python bytecode. <https://docs.python.org/3/library/dis.html>.
- [11] Django patch: Changed the use of `fcntl.flock()` to `fcntl.lockf()`. <https://github.com/django/django/commit/195420259a5286cbeface8ef7d0570e5e8d651e0>.
- [12] Django security releases issued: 2.2.2, 2.1.9 and 1.11.21. <https://www.djangoproject.com/weblog/2019/jun/03/security-releases/>.
- [13] edX. <https://open.edx.org/blog/using-open-edx-ecommerce-module/>.
- [14] Garbage Collector interface. https://docs.python.org/3/library/gc.html#gc.get_objects.

- [15] GlobalInterpreterLock - Python Wiki. <https://wiki.python.org/moin/GlobalInterpreterLock>.
- [16] Gunicorn - Python WSGI HTTP Server for UNIX. <https://gunicorn.org/>.
- [17] Initialization, Finalization, and Threads . https://docs.python.org/3/c-api/init.html#c.PyGILState_Ensure.
- [18] Instrumenting CPython with DTrace and SystemTap. <https://docs.python.org/3/howto/instrumentation.html>.
- [19] Javascript Programming Language. <https://www.javascript.com/>.
- [20] Log the reason for failed PayPal refunds. <https://github.com/pretix/pretix/commit/5400d26c60b7a4fceab2c832419e63abfd785f0d>.
- [21] Long rendered page when a lot of categories products 1910. <https://github.com/django-oscar/django-oscar/issues/1910>.
- [22] Metaobject. <https://en.wikipedia.org/wiki/Metaobject>.
- [23] Multivendor Marketplace Platform - Enterprise Commerce Software. <https://www.shuup.com/>.
- [24] Multivendor Marketplace Platform - Enterprise Commerce Software. <https://www.shuup.com/>.
- [25] Odoo. <https://www.odoo.com/>.
- [26] Odoo Customer Reviews — Success Stories. <https://www.odoo.com/blog/customer-reviews-6>.
- [27] pretix – Reinventing ticket sales for conferences, festivals, exhibitions, ... <https://pretix.eu/about/en/>.
- [28] Product does not get added to basket if force_new_line = True #291. <https://github.com/shuup/shuup/issues/291>.
- [29] Pypy. <https://www.pypy.org/>.
- [30] Ruby Programming Language. <https://www.ruby-lang.org/en/>.
- [31] ruby.git - The Ruby Programming Language. <https://git.ruby-lang.org/ruby.git>.
- [32] The Python programming language. <https://github.com/python/cpython>.
- [33] The Web framework for perfectionists with deadlines — Django. <https://www.djangoproject.com/>.

- [34] What Powers Instagram: Hundreds of Instances, Dozens of Technologies. <https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad>.
- [35] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *The VLDB Journal*, 24(4):557–581, 2015.
- [36] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Data profiling. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1432–1435. IEEE, 2016.
- [37] Ziawasch Abedjan and Felix Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1565–1570, 2011.
- [38] Adam D’Angelo. Why did Quora choose Python for its development? <https://www.quora.com/Why-did-Quora-choose-Python-for-its-development-What-technological-challenges-did-the-founders-face-before-they-decided-to-go-with-Python-rather-than-PHP>, Sep. 2014.
- [39] Alex Martelli. Heavy usage of Python at Google? <https://stackoverflow.com/questions/2560310/heavy-usage-of-python-at-google>, Apr. 2010.
- [40] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: Online patches and updates for security. In *USENIX Security Symposium*, pages 287–302, 2005.
- [41] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.
- [42] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342, 2015.
- [43] Andrew Baumann, Jonathan Appavoo, Robert W Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *USENIX Annual Technical Conference*, pages 337–350, 2007.
- [44] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *USENIX Annual Technical Conference, General Track*, pages 279–291, 2005.
- [45] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. Hitting set enumeration with partial information for unique column combination discovery. *Proceedings of the VLDB Endowment*, 13(12):2270–2283, 2020.

- [46] Nedyalko Borisov and Shivnath Babu. Proactive detection and repair of data corruption: Towards a hassle-free declarative approach with amulet. *Proceedings of the VLDB Endowment*, 4(12):1403–1406, 2011.
- [47] Eric A Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.
- [48] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.
- [49] Alex Bunardzic. Should database manage the meaning? <http://lesscode.org/2005/09/29/should-database-manage-the-meaning/>, 2021.
- [50] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.
- [51] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44, 2006.
- [52] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *29th International Conference on Software Engineering (ICSE’07)*, pages 271–281. IEEE, 2007.
- [53] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 313–325, 2018.
- [54] Christopher John Date. *An introduction to database systems*. Pearson Education India, 1975.
- [55] Django. Constraints reference. <https://docs.djangoproject.com/en/4.0/ref/models/constraints/>, 2021.
- [56] Django. `get()` in `django/db/models/query.py`. <https://github.com/django/django/blob/d8b437b1f3bf54822833bea5e19d2142cf3e1f/django/db/models/query.py#L499>, 2021.
- [57] Django. The web framework for perfectionists with deadlines — django. <https://www.djangoproject.com/>, 2021.
- [58] Django. QuerySet API reference. <https://docs.djangoproject.com/en/4.0/ref/models/querysets/>, Jan. 2022.
- [59] Django. Queryset api reference - `get_or_create()`. <https://docs.djangoproject.com/en/4.0/ref/models/querysets/#get-or-create>, Nov. 2021.

- [60] Django. Writing database migrations. <https://docs.djangoproject.com/en/3.2/howto/writing-migrations/>, Nov. 2021.
- [61] Django-oscar. Change type and name of basket field on abstractorder. <https://github.com/django-oscar/django-oscar/commit/9fa2589b6c70d1f3bff381233eddc41a63aa22e4>, 2021.
- [62] Django-oscar. Check for existing email when updating profile. <https://github.com/django-oscar/django-oscar/pull/324>, 2021.
- [63] Django-oscar. Why is order.basket_id not a foreignkey? https://groups.google.com/g/django-oscar/c/M0FgIB_f9tM/m/W-52L1zZMxAJ, 2021.
- [64] Django-oscar. Make attribute codes unique per product class. <https://github.com/django-oscar/django-oscar/pull/3823>, 2022.
- [65] Django-oscar. Should orderdiscount.offer_id and voucher_id be foreignkey? <https://github.com/django-oscar/django-oscar/issues/3821>, 2022.
- [66] Edx. What makes the open edx platform unique? see these cases. <https://openedx.org/blog/what-makes-open-edx-platform-unique-see-these-cases/>, 2022.
- [67] Edx-commerce. Adding e-commerce to the open edx platform. <https://github.com/openedx/ecommerce/tree/27e6b06b>, 2022.
- [68] Instagram Engineering. Web Service Efficiency at Instagram with Python - Instagram Engineering. <https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366>, 2016.
- [69] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, pages 213–224, 1999.
- [70] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.
- [71] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1001–1012. IEEE, 2018.
- [72] Python Software Foundation. The python profilers. <https://docs.python.org/3.5/library/profile.html>, 2019.
- [73] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 279–292, New York, NY, USA, 2013. Association for Computing Machinery.

- [74] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. *SIGPLAN Not.*, 48(4):279–292, March 2013.
- [75] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
- [76] Google. Google site reliability engineering book. <https://sre.google/sre-book/data-integrity/>, 2022.
- [77] Quinta group. Python at google. <https://quintagroup.com/cms/python/google>.
- [78] gunicorn. Potential http response splitting vulnerability. <https://github.com/benoitc/gunicorn/issues/1227>, 2016.
- [79] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software engineering*, 22(2):120–131, 1996.
- [80] David Heinemeier Hansson. Choose a single layer of cleverness. https://dhh.dk/arc/2005_09.html, 2021.
- [81] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(2):175–204, 1994.
- [82] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, page 249–264, New York, NY, USA, 2012. Association for Computing Machinery.
- [83] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentsch, and Felix Naumann. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment*, 7(4):301–312, 2013.
- [84] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01*, page 13–23, New York, NY, USA, 2001. Association for Computing Machinery.
- [85] Gisli Hjalmtysson and Robert Gray. Dynamic c++ classes-a lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference*, volume 98, 1998.
- [86] IBM. Error and crash recovery from data corruption. <https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=concepts-error-crash-recovery-from-data-corruption>, 2021.

- [87] Information Technology Intelligence Consulting Corp. ITIC 2020 Global Server Hardware, Server OS Reliability Report. <https://www.ibm.com/downloads/cas/DV0XZV6R>, April 2020.
- [88] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 895–913. IEEE, 2016.
- [89] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.
- [90] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.
- [91] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.
- [92] Mirumee Labs. A modular, high performance, headless e-commerce storefront built with python, graphql, django, and reactjs. <https://saleor.io/>, 2020.
- [93] Boyang Li, Denys Poshyvanyk, and Mark Grechanik. Automatically detecting integrity violations in database-centric applications. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 251–262. IEEE, 2017.
- [94] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629, 2016.
- [95] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sharon Lee, Sicheng Pan, Joshua Wu, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. Leveraging application data constraints to optimize database-backed web applications. In *arXiv*, 2022.
- [96] Raymond A Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems (TODS)*, 2(1):91–104, 1977.
- [97] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.
- [98] Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX annual technical conference*, volume 2009. San Diego, CA, 2009.

- [99] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 327–340, 2007.
- [100] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 239–250, 2012.
- [101] Sebastien Martinez, Fabien Dagnat, and Jérémy Buisson. Pymoult : On-Line Updates for Python Programs. In *ICSEA 2015*, 2015.
- [102] Philip Mayer, Michael Kirsch, and Minh Anh Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development*, 5(1):1–33, 2017.
- [103] Jim Melton and Alan R Simon. *SQL: 1999: understanding relational language components*. Elsevier, 2001.
- [104] James Mickens. Rivet: browser-agnostic remote debugging for web applications. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 333–345, 2012.
- [105] MongoDB. Unique constraints on arbitrary fields. <https://www.mongodb.com/docs/manual/tutorial/unique-constraints-on-arbitrary-fields/>, 2022.
- [106] MySQL. How mysql deals with constraints. <https://dev.mysql.com/doc/refman/8.0/en/constraints.html>, 2021.
- [107] Iulian Neamtiu, Michael Hicks, Gareth Stoye, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06*, page 72–83, New York, NY, USA, 2006. Association for Computing Machinery.
- [108] Iulian Neamtiu, Michael Hicks, Gareth Stoye, and Manuel Oriol. Practical dynamic software updating for c. *ACM SIGPLAN Notices*, 41(6):72–83, 2006.
- [109] Joseph P Near and Daniel Jackson. Derailer: interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 587–598, 2014.
- [110] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
- [111] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, 2008.

- [112] NVD. Cve-2018-1000164 detail. https://owasp.org/www-community/attacks/HTTP_Response_Splitting, 2018.
- [113] NVD. Cve-2018-1000164 detail. <https://nvd.nist.gov/vuln/detail/CVE-2018-1000164>, 2018.
- [114] odoo. [payment_paypal] 500 error when process order. <https://github.com/odoo/odoo/issues/39406>, 2019.
- [115] odoo. [13.0] performance issue on validating receipts. <https://github.com/odoo/odoo/issues/46900>, 2020.
- [116] Openedx. Open edx – deliver inspiring learning experiences on any scale. <https://github.com/openedx/edx-platform/tree/97edc47>, 2022.
- [117] Oracle. Oracle database - database concepts - 7 data integrity. <https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-integrity.html>, 2021.
- [118] Oscar. Oscar - domain-driven e-commerce for django. <https://github.com/django-oscar/django-oscar/tree/18c87e>, 2022.
- [119] Oscar. Unique constraints for several table’s columns. <https://github.com/django-oscar/django-oscar/pull/3868>, 2022.
- [120] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment*, 8(7):774–785, 2015.
- [121] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP ’09*, page 87–102, New York, NY, USA, 2009. Association for Computing Machinery.
- [122] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 573–585. ACM, 2019.
- [123] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA ’14*, page 103–119, New York, NY, USA, 2014. Association for Computing Machinery.
- [124] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, 2003.

- [125] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, 2002.
- [126] PostgreSQL. 11.8. partial indexes. <https://www.postgresql.org/docs/current/indexes-partial.html>, 2021.
- [127] PostgreSQL. 5.4. constraints chapter 5. data definition. <https://www.postgresql.org/docs/current/ddl-constraints.html>, 2021.
- [128] Python. ast — abstract syntax trees. <https://docs.python.org/3/library/ast.html>, 2022.
- [129] Rails. Active record migrations. https://edgeguides.rubyonrails.org/active_record_migrations.html, 2021.
- [130] Rails. Concurrency and integrity for uniqueness in rails. https://github.com/rails/rails/blob/main/activerecord/lib/active_record/validations/uniqueness.rb#L179, 2021.
- [131] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.
- [132] Romain Komorn. Python in production engineering. <https://engineering.fb.com/production-engineering/python-in-production-engineering/>, May 2016.
- [133] Florian Rommel, Christian Dietrich, Peng Huang, Daniel Friesel, Sangeetha Abdu Jyothi, Karan Grover, Marcel Köppen, Nina Narodytska, Muthian Sivathanu, Christoph Borchert, et al. From global to local quiescence: Wait-free code patching of multi-threaded processes. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 651–666, 2020.
- [134] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *WebDB*, 2009.
- [135] saleor. Category index renders extremely slow when there are many discounts. <https://github.com/mirumee/saleor/issues/1314>, 2017.
- [136] Saleor. Error in the dashboard. <https://github.com/saleor/saleor/issues/1670>, 2021.
- [137] Saleor. Make order.total not nullable. <https://github.com/saleor/saleor/pull/1893>, 2021.
- [138] Saleor. Saleor – a headless, graphql-first, open-source e-commerce platform. <https://github.com/saleor/saleor/tree/53e519df>, 2021.
- [139] Saleor. Rebooting enterprise with open source. <https://saleor.io/enterprise-open-source/>, 2022.

- [140] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.
- [141] Shuup. Multivendor marketplace platform - enterprise commerce software. <https://github.com/shuup/shuup/tree/25f78c>, 2022.
- [142] similartech. Market share and web usage statistics of Django. <https://www.similartech.com/technologies/django>, Jan. 2020.
- [143] Craig AN Soules, Jonathan Appavoo, Kevin Hui, Robert W Wisniewski, Dilma Da Silva, Gregory R Ganger, Orran Krieger, Michael Stumm, Marc A Auslander, Michal Ostrowski, et al. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track*, pages 141–154, 2003.
- [144] Stackexchange. Why are constraints applied in the database rather than the code? <https://dba.stackexchange.com/questions/39833/why-are-constraints-applied-in-the-database-rather-than-the-code>, 2021.
- [145] Stackoverflow. Should you enforce constraints at the database level as well as the application level? <https://stackoverflow.com/questions/464042/should-you-enforce-constraints-at-the-database-level-as-well-as-the-application>, 2021.
- [146] Statista Inc. Average cost per hour of enterprise server downtime worldwide. <https://www.statista.com/statistics/753938/worldwide-enterprise-server-hourly-downtime-cost/>, 2021.
- [147] Guy Steele. *Common LISP: the language*. Elsevier, 1990.
- [148] Gareth Stoye, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05*, page 183–194, New York, NY, USA, 2005. Association for Computing Machinery.
- [149] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.
- [150] Suriya Subramanian, Michael Hicks, and Kathryn S McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2009.
- [151] Davy Suvéé, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, 2003.

- [152] Wei Tang and Min Zhang. Pyreload: Dynamic updating of python programs by reloading. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 229–238. IEEE, 2018.
- [153] Teradata. Teradata - physical database integrity. <https://docs.teradata.com/r/sUbveBFyhttlbZzLz7nJLw/x~5k~cGbb5CIg7WrWgBNoA>, 2021.
- [154] thenewstack.io. Instagram Makes a Smooth Move to Python 3. <https://thenewstack.io/instagram-makes-smooth-move-python-3/>, Jun. 2017.
- [155] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user’s site. *ACM SIGOPS Operating Systems Review*, 41(6):131–144, 2007.
- [156] Gerd Wagner. Chapter 9. implementing constraint validation in a java ee web app. <https://web-engineering.info/book/WebApp1/ch09.html>, 2021.
- [157] Wagtail. Wagtail cms: Django content management system. <https://github.com/wagtail/wagtail/tree/317f10>, 2022.
- [158] Yuepeng Wang, Isil Dillig, Shuvendu K Lahiri, and William R Cook. Verifying equivalence of database-driven applications. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.
- [159] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, 2019.
- [160] Todd Warszawski and Peter Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 5–20, 2017.
- [161] Wikipedia. Youtube. <https://en.wikipedia.org/wiki/YouTube>, 2020.
- [162] Cong Yan and Alvin Cheung. Leveraging lock contention to improve oltp application performance. *Proceedings of the VLDB Endowment*, 9(5):444–455, 2016.
- [163] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. *ACM SIGPLAN Notices*, 51(6):631–647, 2016.
- [164] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1098–1109. IEEE, 2020.
- [165] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3(1-2):805–814, 2010.

- [166] Zulip. Corrupted reactions data model state when using multiple aliases for an emoji code. <https://github.com/zulip/zulip/issues/15347>, 2021.
- [167] Zulip. migrations: Add case-insensitive unique indexes on realm and email. <https://github.com/zulip/zulip/commit/b9b146c8095648d4ef61650a89bfe6f557308574>, 2021.
- [168] Zulip. psycopg2.errors.uniqueviolation duplicate key value violates unique constraint. <https://github.com/zulip/zulip/issues/15772>, 2021.
- [169] Zulip. Case study: Rust programming language community. <https://zulip.com/case-studies/rust/>, 2022.
- [170] Zulip. Change realm field to be not null in attachment. <https://github.com/zulip/zulip/pull/21470>, 2022.
- [171] Zulip. Realm field in the realmauditlog table. <https://chat.zulip.org/#narrow/stream/9-issues/topic/realm.20field.20in%20table%20RealmAuditLog%20and%20RealmUserDefault/near/1335322>, 2022.
- [172] Zulip. Unique constraints for several table's columns. <https://chat.zulip.org/#narrow/stream/9-issues/topic/Several.20generated.20key.20fields.20w.20Fo.20unique%20constraints.2E>, 2022.
- [173] Zulip. Zulip. <https://github.com/zulip/zulip/tree/f5bb43ab>, 2022.