It's About Time: An Empirical Study of Date and Time Bugs in Open-Source Python Software

Shrey Tiwari
Carnegie Mellon University
Pittsburgh, PA, USA
shrey@cmu.edu

Serena Chen*
University of California, San Diego
San Diego, CA, USA
sec022@ucsd.edu

Alexander Joukov*
Stony Brook University
Stony Brook, NY, USA
ajoukov@cs.stonybrook.edu

Peter Vandervelde*
University of California, Santa Barbara
Santa Barbara, CA, USA
pvandervelde@ucsb.edu

Ao Li
Carnegie Mellon University
Pittsburgh, PA, USA
aoli@cmu.edu

Rohan Padhye
Carnegie Mellon University
Pittsburgh, PA, USA
rohanpadhye@cmu.edu

***Abstract*—Accurately performing date and time calculations in software is non-trivial due to the inherent complexity and variability of temporal concepts such as time zones, daylight saving time (DST) adjustments, leap years and leap seconds, clock drifts, and different calendar systems. Although the challenges are frequently discussed in the grey literature, there has not been any systematic study of date/time issues that have manifested in real software systems. To bridge this gap, we qualitatively study 151 bugs and their associated fixes from open-source Python projects on GitHub to understand: (a) the *conceptual categories* of date/time computations in which bugs occur, (b) the *programmatic operations* involved in the buggy computations, and (c) the underlying *root causes* of these errors. We also analyze metrics such as bug severity and detectability as well as fix size and complexity.**

Our study produces several interesting findings and actionable insights, such as (1) time-zone-related mistakes are the largest contributing factor to date/time bugs; (2) a majority of the studied bugs involved incorrect construction of date/time values; (3) the root causes of date/time bugs often involve misconceptions about library API behavior, such as default conventions or nuances about edge-case behavior; (4) most bugs occur within a single function and can be patched easily, requiring only a few lines of simple code changes. Our findings indicate that static analysis tools can potentially find common classes of high-impact bugs and that such bugs can potentially be fixed automatically. Based on our insights, we also make concrete recommendations to software developers to harden their software against date/time bugs via automated testing strategies.

***Index Terms*—date, time, time zone, DST, empirical study, software bugs**

I. Introduction

Time is a fundamental concept in software engineering. Whether it's scheduling flights, processing bank transactions, computing payroll wages, validating digital certificates, controlling industrial processes, or logging operational data, the correctness of date and time logic is of utmost importance across a multitude of systems and functions.

Accurately performing date and time calculations in software is non-trivial due to the inherent complexity and variability of temporal concepts such as time zones, daylight saving time (DST) adjustments, leap years and leap seconds, clock drifts, and different calendar systems. Software often needs to handle dates and times across various geographical regions, each with its own conventions and peculiarities.

Software bugs relating to date and time logic can lead to accidental misinformation [1], data corruption [2], estimation errors [3], security breaches [4], financial losses [5], and even legal liabilities [6]. For example, a bug in the systems of the US Patent and Trademark Office assigned legally inaccurate expiration dates to over 27,000 patents [6], potentially exposing hundreds of millions of dollars to litigation. Electronic health records in a widely used medical software system in the U.S. have been reported to get corrupted during daylight savings transitions, leading hospitals to suspend operations or switch to paper twice a year [2]. In a 2007 incident, the U.S. Air Force's F-22 Raptors experienced navigational malfunctions when crossing the international date line [7]. In 2024, some gas stations in New Zealand failed to dispense fuel on February 29th [5]. These examples underscore the importance of date and time logic in maintaining the integrity and reliability of critical software systems.

To alleviate some of these challenges, programming languages usually offer standard libraries to support date/time computations. Often, third-party libraries provide alternative methods for representing and manipulating temporal data. The design choices of each library—which may affect correctness in edge cases—are typically identifiable only through natural-language API documentation or the implementation code itself. Inconsistencies between conventions used across libraries and programming languages can lead to bugs when developers interoperate between multiple APIs. Although these practical challenges with programming data/time logic are frequently discussed in the *grey literature* [8, 9, 10], there is no empirical data about the common or most important types of date/time issues that developers face. We aim to bridge this gap.

In this paper, we present a systematic qualitative study of date and time bugs occurring in open-source GitHub projects

*These authors contributed equally to the paper.

using the Python programming language. Specifically, we manually analyze *151 date/time bugs* and their associated fixes to understand root causes and identify common patterns. We use an open-coding approach to classify bugs along three dimensions: (a) the *conceptual categories* of date/time computations (such as time zones and DST) in which the bug occurred, (b) the *programmatic operations* involved in the buggy computations, and (c) the underlying *root causes* of these errors, as evidenced by the associated fixes. We also qualitatively analyze four metrics: bug severity, bug detectability, fix size, and fix complexity.

Our study produces several interesting findings and actionable insights, such as (1) time-zone-related mistakes are the largest contributing factor to date/time bugs; (2) a majority of the studied bugs involved incorrect construction of date/time values; (3) the root causes of date/time bugs often involve misconceptions about library API behavior, such as default conventions or nuances about edge-case behavior; (4) most bugs occur within a single function and can be patched easily, requiring only a few lines of simple code changes.

Our findings indicate that there is potential for static analysis tools to discover common classes of high-impact bugs automatically. We demonstrate this potential via a small case study using CodeQL. We also recommend strategies for software developers to harden their software against date/time bugs via automated testing: specifically, using a combination of time-mocking tools and increasing test coverage for discovering shallow time-zone-related bugs; using property-based testing for string-formatting/parsing; and using fuzz testing for uncovering edge-case DST and out-of-bounds issues.

To summarize, this paper makes the following contributions:

1) We present the first systematic study of date and time bugs occurring in open-source software and extract several insights based on a multi-dimensional qualitative analysis (Section IV).
2) Based on our insights, we provide concrete recommendations to developers for strengthening current software and also highlight the potential for further research using static analysis for automatic bug detection (Section V).
3) We make our manually analyzed and fully annotated dataset of 151 bugs and their associated fixes public to promote further research on this topic (Section VII).

II. Background

A. Motivating Example—Why Date and Time is Hard

Consider a software developer attempting to perform a simple task: *writing a Python program that calculates the number of hours until 5 pm tomorrow local time*. They might think of doing so via a series of reasonable steps: (a) get the current local time via `datetime.now()`; (b) get the target time by incrementing the date by one and setting the clock to 17:00:00; (c) compute the difference between the target and current time, and return a value in hours. Figure 1a shows such an implementation, which we got by querying ChatGPT-4o [11] (other leading AI models produced the same solution).

```python
from datetime import datetime, timedelta

def hours_until_5pm_tomorrow():
  # Get the current date and time
  current_time = datetime.now()

  # Calculate the date for tomorrow
  tomorrow = current_time + timedelta(days=1)

  # Set the time for 5 PM tomorrow
  target_time = datetime(tomorrow.year, tomorrow.month,
                tomorrow.day, 17, 0)

  # Calculate the time difference
  time_diff = target_time - current_time

  # Convert the time difference to hours
  return time_diff.total_seconds() / 3600
```

(a) Almost correct implementation for calculating hours until 5pm the next day. The result is inaccurate when a DST transition occurs between now and 5pm tomorrow.

```python
from datetime import datetime, timedelta, timezone

def hours_until_5pm_tomorrow():
  # Get the current date and time
  current_time = datetime.now() .astimezone()

  # Calculate the date for tomorrow
  tomorrow = current_time + timedelta(days=1)

  # Set the time for 5 PM tomorrow
  target_time = datetime(tomorrow.year, tomorrow.month,
                tomorrow.day, 17, 0, 0) .astimezone()

  # Calculate the time difference
  time_diff = target_time.astimezone(timezone.utc) -
              current_time.astimezone(timezone.utc)

  # Convert the time difference to hours
  return time_diff.total_seconds() / 3600
```

(b) A corrected version requiring time-zone-awareness for addressing the DST issue. When executed on March 9, 2024, at 11pm ET, this code correctly computes *17 hours*.

Fig. 1: Python code to *calculate the number of hours until 5pm tomorrow local time*. The seemingly correct solution in 1a was authored by OpenAI's GPT-4o (variable names truncated). The truly correct version in 1b is more nuanced; accurately performing date/time computations is hard!

It turns out that this implementation is correct *most of the time*; however, there is a critical edge-case bug. If the code is executed in a location that observes daylight saving time (DST) transitions, and such a transition occurs between now and 5 pm tomorrow, the code will return an incorrect result! For example, if this code were to be executed in New York on March 9, 2024, at 11 pm, then the code would return "18" whereas there are only 17 hours left until 5 pm the next day due to the DST transition where clocks "spring forward" by one hour at 02:00. The problem is that the time differencing operator cannot account for DST changes when time zone information is missing. Knowing this, one may consider performing all computations in UTC; however this cannot be the solution either—the current date in UTC would be "March 10th", so adding one day to a UTC timestamp

would also lead to incorrect results.

The correct solution requires performing time-zone-aware arithmetic. Shown in Figure 1b is a solution that attempts this by (a) using a representation tagged with the local time zone to correctly perform calendar arithmetic (Lines 23 and 30), and (b) using a representation in UTC to correctly perform time differencing (Line 33). In general, time arithmetic (e.g., determining if A is within X hours/minutes/seconds of B) is best done using a fixed reference (e.g., using a representation like *Unix time*, which counts the number of seconds since an epoch in UTC); whereas date arithmetic (e.g., determining if A is within Y years/months/days of B) is best done using a calendar-and-time-zone-aware representation.

Though Figure 1a showcases one specific flawed implementation, many pitfalls can arise when computing durations until a target time. Developers may neglect to use operators that account for DST transitions, inadvertently mix naive and time-zone-aware `datetime` objects, or err when constructing new time-zone-aware `datetime` instances—potentially creating non-existent times that Python still considers valid. Our empirical study is motivated by the need to understand the prevalence of these issues in open-source software.

B. Current Landscape

Python offers a range of libraries for date and time computation, starting with its built-in `datetime` [12] library, which provides support through various modules such as `datetime`, `zoneinfo`, `time`, and `calendar`, and types including `date`, `time`, `datetime`, `tzinfo`, `timedelta`, and `timezone`. Despite its comprehensive support, developers often find the standard library unintuitive and hard to use correctly [10, 13].

To address these challenges, several third-party libraries have been developed. *Pendulum* [14] is one such library that builds upon the standard library, claiming to offer cleaner and more intuitive APIs. Another popular option is *Arrow* [15], which not only provides standard date and time operations but also supports so-called humanized representations (e.g., "2 days ago") across numerous locales. The *Whenever* [16] package provides an API that uses the (dynamic) type system to enable time zone/DST-safe calculations. Libraries like *Dateutil* [17] and *Pytz* [18] offer powerful time zone extensions to the standard library. Beyond these, there are several other specialized libraries, such as *Heliclockter* [19], *Maya* [20], *Delorean* [21], *Moment* [22], and *Chronyk* [23], each introducing unique features and enhancements. While many of these libraries are designed to interoperate with the standard library, differences in usage patterns, operator semantics, and error handling can be a source of confusion for developers.

III. Methodology

Open-source GitHub projects often report code bugs through GitHub issues, which provide valuable information for our empirical study. These issues contain details such as descriptions, reproducibility steps, related issues, developer discussions, and code-level fixes. We compiled a comprehensive dataset of GitHub issues specifically related to date/time bugs, with all data collected from GitHub before July 9th, 2024.

A. Data Collection Process

Our research examines the landscape of date/time bugs in popular Python projects by systematically mining and filtering relevant information from GitHub, as detailed below.

Repositories Selection: Our study considered repositories that: (a) were created after 2014, (b) have over 100 stars, and (c) primarily use Python for code and English for communication. Applying these filters using the GitHub GraphQL API [24] resulted in *55,585 repositories*.

We then focused on repositories performing temporal computations, specifically those importing at least one of these date/time libraries: *Datetime*, *Arrow*, or *Pendulum*. Each library has over 300 stars and is used by at least 10 repositories identified earlier. We excluded lesser-known or infrequently used libraries like *Maya*, *Delorean*, *Moment*, *Whenever*, *Heliclockter*, and *Chronyk*. This filtering resulted in *22,132 repositories* using widely adopted Python date/time libraries.

Keywords: To uncover date and time bugs, we compiled a comprehensive list of keywords capturing relevant concepts across various dimensions. These keywords guided our search for potential date/time issues.

- **Module Names:** *datetime*, *arrow*, *pendulum*, *dateutil*, *pytz* (the last 2 are add-ons used with *datetime*)
- **Units of Time:** *nanosecond*, *millisecond*, *microsecond*, *second(s)*, *day(s)*, *week(s)*, *month(s)*, *year(s)*, *epoch*
- **Python Methods:** *strptime*, *strftime*, *timestamp*, *utcnow*, *fromtimestamp*, *localtime*, *timedelta*
- **Date and Time Concepts:** *timezone(s)*, *gmt*, *utc*, *dst*, *daylight*, *fold*, *leap*
- **Miscellaneous:** *elapsed*, *interval(s)*, *duration(s)*

Issue Compilation and Filtering: Utilizing the GitHub GraphQL API, we scraped issues from each repository in our dataset that included at least one of the aforementioned keywords. This resulted in *26,967 issues*, capturing their titles, developer discussions, pull request and commit links, and tags.

However, not all of these issues are pertinent to our study. Some may not be confirmed bugs, lack sufficient information, or be unrelated to temporal computations. Therefore, we refined our dataset using these filtering criteria:

- **Bug-Related Terms:** The issue's title or tags must include "bug", "fix", or "wrong", indicating it's a bug rather than a query or suggestion.
- **Exclusion of Feature Requests:** The issue must not be labeled "enhancement" or "feature", which typically denote new feature requests.
- **Closed Issues:** The issue must be closed on GitHub, confirming it has been resolved.
- **Adequate Information:** The issue must contain sufficient information for analysis. We ensure this via a minimum character limit for the combined content of the issue title, description, and discussions. Through manual experimentation with character limits of 250, 500, and 1,000

characters, we found that a threshold of 1,000 characters offered the optimal balance and thus selected it for our study.

- **Referenced in Commits or Pull Requests:** The issue must be referenced in a commit or pull request, allowing us to examine the resolved code to identify the bug's root cause.

B. Bug Selection

Following the final filtering step, we identified *9,669 issues*. While this dataset was more manageable, it still posed challenges for manual analysis. Not all issues were useful for our study, as some were unrelated to date/time bugs, duplicate, too complex, or had fixes in non-Python code—we term them as *false positives*. Automatic elimination of these is not feasible and manual filtering is time-consuming; so, we opt for a strategy to use statistical heuristics to rank issues and then perform manual filtering on a set of high-quality results.

To efficiently prioritize and categorize issues for detailed analysis, we utilize Term Frequency-Inverse Document Frequency (TF-IDF) [25], a key concept in information retrieval. TF-IDF ranks documents by assessing the relevance of terms based on their frequency in a document (term frequency) and their uniqueness across the document collection (inverse document frequency). This ranking allows our manual false positive filtering step to be more successful on average.

First, we computed the inverse document frequency (IDF) for each of our keywords on a representative corpus of issues created by selecting 1,000 popular Python repositories and gathering the 100 most recent issues from each. Then, for each issue in our scraped dataset of 9,669 issues, we compute the TF score for each of our keywords and combine it with the precomputed IDF score, giving us the TF-IDF score for that issue.

The TF-IDF metric prioritizes issues with multiple keywords that are present more often in date/time-related bugs than general software issues. The insight here is that date/time bugs contain multiple distinguishing keywords (keywords with high IDF scores) from our keywords list, hence receiving a higher TF-IDF score, as compared to issues that are not date/time bugs. For example, an issue on performance bugs containing the phrase "elapsed time" will be pulled in our scraping step. However, it is unlikely to contain many other keywords from our list, resulting in a low issue TF-IDF score and being deprioritized.

After computing the TF-IDF scores, we sorted the issues in descending order and incrementally reviewed them in batches. Note that these batches still contained false positives, which we pruned as part of the manual process described next.

C. Classification Strategy

As demonstrated in our motivating example (Section II-A), date/time bugs can be particularly challenging. Moreover, issues with vastly different root causes and effects can exhibit similar symptoms (e.g., an off-by-one error in dates/hours). To capture these intricacies, we adopt a hierarchical labeling approach. First, from a *requirements* point of view, we identify the date/time concept involved in the computation that turned out to be buggy. We then identify the code-level programmatic operations (e.g., Python APIs) that were being used incorrectly. Lastly, recognizing that there are multiple ways of misusing date/time APIs, we identify the type of mistake made by the developer in the computation. This hierarchical approach—identifying the conceptual gap, the specific erroneous computation, and the human error—helps us capture a multifaceted view of complex date/time bugs. We also qualitatively assess whether the bug is obvious or an obscure edge case, its impact on the software system, and the complexity of fixing it. In total, we identified *seven factors* for this study, detailed in Table I.

TABLE I: Factors studied for the date/time bugs.

Study Factor	Description
Conceptual Category	Based on the issue description, what are the date/time concepts that are involved in the problem developers were trying to solve?
API/Coding Category	What are the programmatic operations in the Python code where the bug manifests?
Root Cause	Based on the fixing patch, what was the mistake or misconception that led to the bug?
Bug Obscurity	How challenging is it for a developer to find this bug during development, testing, or production runs?
Bug Severity	What is the impact of the bug on the software that runs the buggy code?
Fix Size	How big is the code change required to fix the bug?
Fix Complexity	How logically complex is the code change required to fix the bug?

We employed an open coding approach [26] to develop a taxonomy for the factors *Conceptual Category*, *API/Coding Category*, and *Root Cause*. For the factors *Bug Obscurity*, *Bug Severity*, *Fix Size*, and *Fix Complexity*, we adopted an ordinal scale with predefined categories: "small", "medium", and "large" (additionally "very large" for *Fix Size* based on observations). Note that we encountered two challenges in objectively counting *Fix Size*: (1) Pull Requests (PRs) often contain multiple commits, not all related to the bug; (2) Individual commits can include changes not directly related to the bug, such as code refactoring, fixes to other bugs, test code updates, comment modifications, documentation updates, and import changes. Simply counting the lines of code (LoC) modified in the PRs would incorporate this noise. Therefore, we opted for a manual review to subjectively categorize the amount of changes needed for fixing the bug.

Sample Classification: To understand our analysis process, let's analyze the motivating example from Section II-A. The program in Figure 1a attempts to compute a duration but produces an incorrect result when the duration overlaps with a DST transition because the `datetime` subtraction operator does not account for DST changes. We classify this bug under the concepts "DST" and "Duration" (full list in Fig. 3). The fixed version of this program shown in Figure 1b

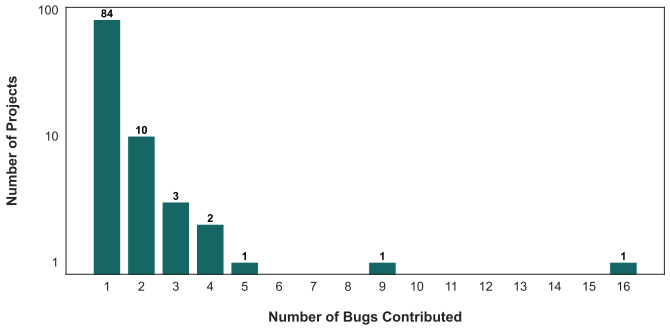


Fig. 2: Distribution of projects selected in our study, showcasing the number of bugs identified per project. The plot illustrates that the majority of projects contribute a single bug, with two outlier projects contributing *9* (*Dateutil*) and *16* (*Pendulum*) bugs.

reveals that original mistakes occurred when dealing with the programmatic operations for "`datetime` construction" and "date and time arithmetic" (full list in Fig. 4). Ultimately, the root cause of the bug was failing to perform time-zone-aware computations or, in other words, "using naive `datetime` objects incorrectly" (full list in Fig. 5). The program depicted in Figure 1a silently produces the wrong result, rendering the bug highly obscure. The severity of the issue varies depending on the context in which the computation is executed. Since the resolution involves modifying code within a single function and performing multiple timezone conversions, the fix is categorized as moderate in both size and complexity. The complete taxonomy is described in Section IV.

The program depicted in Figure 1a generates an incorrect result without any explicit indication, rendering the bug particularly elusive.

Study Setup: We began with four members of the research team using an open coding approach to analyze and label issues from our dataset in batches until no new classes emerged. Each batch was subdivided into equal parts and assigned to one of the six possible pairs formed by the four members. Each pair analyzed their assigned bugs and created a list of categories. We then collectively reviewed these categories and agreed upon common definitions and a unified taxonomy for each factor. We observed saturation in our taxonomies after analyzing 200 issues. Next, we created a test set of 60 issues (10 per pair) to check that no new labels emerged.

During the manual analysis process, we filtered out issues that were deemed as false positives (i.e., not date/time bugs). Overall, we analyzed a total of *260* issues originating from *144* unique open-source repositories and classified *151* true date/time-related bugs from *102* distinct projects. Figure 2 illustrates the frequency of projects contributing varying numbers of bugs to our final dataset.

We used the true positives from the test set to measure inter-annotator agreement using Cohen's Kappa [27]. For ordinal data, we applied the weighted Cohen's Kappa to account for the degree of disagreement. If a bug had multiple labels for a factor, we resolved the ambiguity by focusing on the most significant label, which we collectively agreed to assign as the first label.

Result: We observed moderate to substantial agreement for each of the factors[1]: Conceptual Category (0.65), API/Coding Category (0.68), Root Cause (0.57), Bug Obscurity (0.49), Bug Severity (0.56), Fix Size (0.66), and Fix Complexity (0.53).

D. Threats to Validity

Selection Criteria: Our selection criteria focused on analyzing bugs that were closed and contained adequate information. This could pose a threat to validity since we exclude open or ongoing bugs. However, concentrating on real-world fixed bugs enables us to draw concrete conclusions about the study factors and this approach aligns with multiple other empirical studies [28, 29, 30, 31].

Representativeness of Selected Bugs: Our study of 151 bugs offers meaningful contributions to our understanding of date/time bugs. However, the limited number of bugs analyzed could threaten external validity. To address this concern, we performed manual analysis until our taxonomy reached saturation. Additionally, to mitigate potential bias stemming from date/time library bugs, we provide visual illustrations of how different project types contribute to each label in our taxonomy. Our time and effort-intensive manual analysis of 151 bugs (requiring $\sim$520 person-hours) aligns well with the standards of state-of-the-art bug studies [28, 29, 31], which have analyzed between 100 and 400 bugs.

Generalization Across Programming Languages: Our study focuses on the Python programming language; this could threaten external validity as our results may not generalize to other languages with varying date/time representations and libraries. For instance, JavaScript's `Date` class and Java's (now outdated) `java.util.Date` classes utilize an epoch-based system, whereas Java's `java.time` classes provide comprehensive API support for both epoch-based and calendar-based date/time representations. These contrast with Python's default calendar-based `datetime` module and the differences in internal representations may give rise to unique bug patterns and distributions. Although our results may not be universally applicable, our study remains valuable as it offers meaningful insights into date/time bugs within the ecosystem of the most popular programming language [32].

Manual Analysis: Our manual analysis of bugs acknowledges the possibility of misclassifications due to ambiguity or inherent complexity of date/time bugs. To mitigate this, we ensured that any labeling disagreements among the research team members were effectively resolved through collaborative discussions until a consensus was achieved.

IV. Results

In this section, we present the results of our bug analysis and the taxonomy we developed for each of the seven aforementioned study factors.

[1]For each factor, we calculate the inter-annotator agreement scores for all annotator pairs and then report averaged scores.

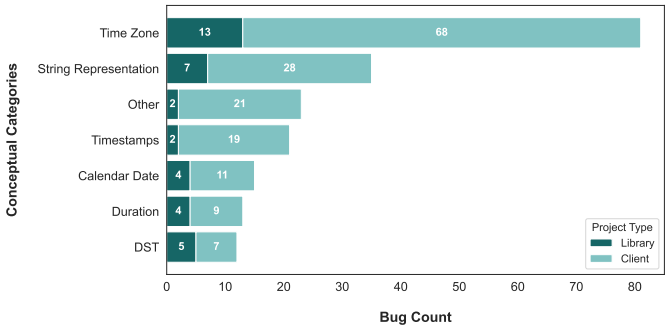


Fig. 3: Distribution of bugs across identified conceptual categories. The plot reveals that problems related to time zones are the biggest contributors to bugs.

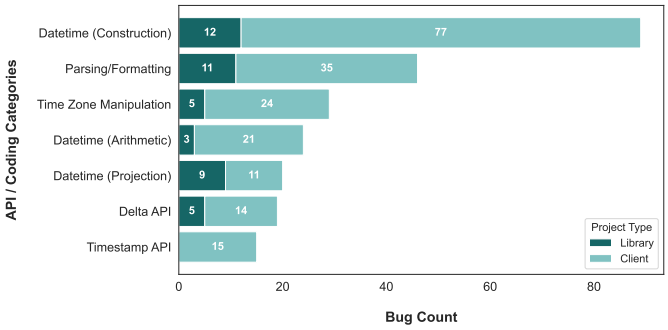


Fig. 4: Distribution of bugs across identified API/coding categories. The plot indicates that inaccurate construction of `datetime` objects is the most frequent source of bugs.

Our dataset includes bugs from two sources: client projects that utilize date/time libraries for temporal calculations and the date/time libraries themselves ("*Dateutil*", "*Pendulum*", "*Maya*", "*Chronyk*", and "*Arrow*"). To illustrate the distribution of bugs, we present stacked bar charts that delineate the contributions of each project type.

A. What date/time concepts do we observe in the bugs?

Figure 3 illustrates the distribution of bugs across seven distinct date/time concept categories we identified by analyzing only their *issue descriptions*. We next describe these in detail.

Time Zone: Approximately *53.6%* of the bugs are associated with time zones. Errors can arise while: displaying date/time information localized to a specific time zone (e.g., twintproject/twint#401), working with task/job scheduling (e.g., apache/airflow#29576 and PrefectHQ/prefect#1053; in the latter, a "test [...] fails whenever it's run between 9-9:30AM EST"), or when working with relative dates (e.g., kennethreitz/maya#36, which incorrectly computes "tomorrow" for a UTC value without considering the local calendar date).

String Representation: *23.2%* of bugs occur when formatting date/time information for printing or writing to a database (e.g., googleapis/python-bigquery#392, where precision is lost when inserting a specific timestamp to Google BigQuery) or parsing input date and time information for various tasks (e.g., pyopenapi/pyswagger#83, which drops milliseconds from input strings, and brython-dev/brython#1849, which could not parse double-digit month values).

Timestamps: Python supports epoch-based date and time representation accurate to the microsecond. *13.9%* of bugs are related to timestamps and arise when developers convert timestamp information from one time zone to another (e.g., holoviz/holoviews#2459, ranaroussi/yfinance#545) or interpret timestamp information incorrectly (e.g., aimhubio/aim#1084 and googleapis/python-bigquery-pandas#261, both of which incorrectly determine whether a stored timestamp is UTC).

Calendar Date: *9.9%* of the bugs are related to calendar dates. Bugs that deal with only dates can arise while handling historical data (e.g., SEED-platform/seed#1439 related to buildings built before 1900) or ambiguous/incomplete values (e.g., parsing the year "85" as in sdispater/pendulum#686), or while dealing with ordinal days of the year/month/week (e.g., sktime/sktime#3188, which corrupted time-series data).

Duration: *8.6%* of bugs are related to duration computations such as dealing with periodic intervals (e.g., stringertheory/traces#217, which produces an unexpected time series) or performing arithmetic (e.g., thombashi/DateTimeRange#44, which deals with subtracting negative durations from a date).

DST: only *7.9%* of bugs are due to DST changes. On certain dates, specific time zones may experience instances where certain times either do not exist or occur twice. It is crucial for programs to handle these situations gracefully, as failing to do so can result in crashes or, even worse, silent errors. DST can affect any kind of arithmetic computations on date/time objects like task scheduling (e.g., kiorky/croniter#56) or calendar manipulations (e.g., sdispater/pendulum#768).

Other: About *15.2%* of bugs in our study are general code bugs not specific to any date/time concept. For example, bugs that arise due to deprecated date/time APIs (e.g, googleapis/python-storage#1194) or hard-coded return types (e.g., sdispater/pendulum#203) are tagged with this label.

> 💡 **Insight #1:** Time zones represent the most significant source of bugs in our study (*53.6%*), indicating they are a common pain point for developers when performing date/time computations. Interestingly, while frequently discussed in grey literature [2, 10, 33, 34], only a small fraction (*7.9%*) of bugs we studied are related to DST.

B. What programmatic operations are involved in the buggy Python code?

Figure 4 illustrates the seven distinct categories of programmatic operations that we identified as contributors to date/time bugs in our dataset. We explore these in detail below:

`Datetime` (Construction): A majority of the bugs, approximately *58.9%*, fall into this category. Bugs can arise when one creates `datetime` objects in the local time zone using `datetime.now()` and then misinterprets them as belonging to the UTC time zone (e.g., googleapis/python-storage#244).

Bugs can also arise when one incorrectly calls the constructor for `datetime` objects, `datetime.datetime()`. For example, in mverleg/pyjson_tricks#41, the *Pytz* time zone API is incorrectly being used in the constructor leading to a time-zone-aware object being created with the wrong time zone information.

Parsing/Formatting: *30.5%* of bugs were related to incorrect handling of string representations. More specifically, *86.9%* of these bugs were due to errors while handling programmatic string formats. For example, a parsing error occurs in kayak/pypika#152 due to an error in the regex being used leading to trailing zeros in intervals to be completely omitted (e.g., 100 seconds was parsed as 1 second)! Bugs can also arise while formatting or serializing data to strings. For example, in skarim/vobject#32 an error occurs when trying to serialize a non-existent `datetime`. The remaining *13.1%* of the bugs arise while trying to handle non-standard humanized string formats. For example, in KoffeinFlummi/Chronyk#5, a day out-of-range error is thrown when parsing the humanized string "in 4 months" or "3 months ago" on the 31st of May.

Time Zone Manipulation: *19.2%* of the bugs arise while trying to convert date/time information from one time zone to another (e.g., rpy2/rpy2#634) or while comparing distinct Python objects that represent the same logical time zone (e.g., dateutil/dateutil#151 or pydantic/pydantic#8683).

`Datetime` (Arithmetic): *15.9%* of bugs arise while performing arithmetic operations on date/time information. For example, in agronholm/apscheduler#911 due to incorrect calculations, every alternate task trigger is skipped. Another example is sdispater/pendulum#768 where, due to a DST-related corner case, adding one day to a `datetime` object does not change the date but rather adds 23 hours!

`Datetime` (Projection): *13.2%* bugs occur while operating on individual components of a `datetime` object. For example, in sdispater/pendulum#152 there is an off-by-one-second error due to an incorrect choice for the internal representation of seconds in a `datetime` object. Similarly, a `NoneType` object-dereference error arises in dateutil/dateutil#132 when the code tries to access the time zone information of a `time` object (time objects do not have time zone information associated to them).

Delta API: About *12.6%* of bugs deal with computations involving durations. For example, due to a calculation error in sdispater/pendulum#475, adding a duration of one month to other `duration` objects has no effect. Another example is model-bakers/model_bakery#322, where a type mismatch error is raised when trying to add a `timedelta` to a time-zone-aware `datetime` object.

Timestamp API: *9.9%* of bugs fall into the last category of our taxonomy. These are bugs that arise while performing computations with timestamps. Examples include dora-metrics/pelorus#662 where conversion of `datetime` objects to timestamps fails in the presence of time zone information. Similarly, in python-social-auth/social-core#128 the expiration duration calculations produce a wrong result due to mixing up of UTC localized `datetime` objects with `datetime` objects returned by `datetime.fromtimestamp()` which use local time zone information.

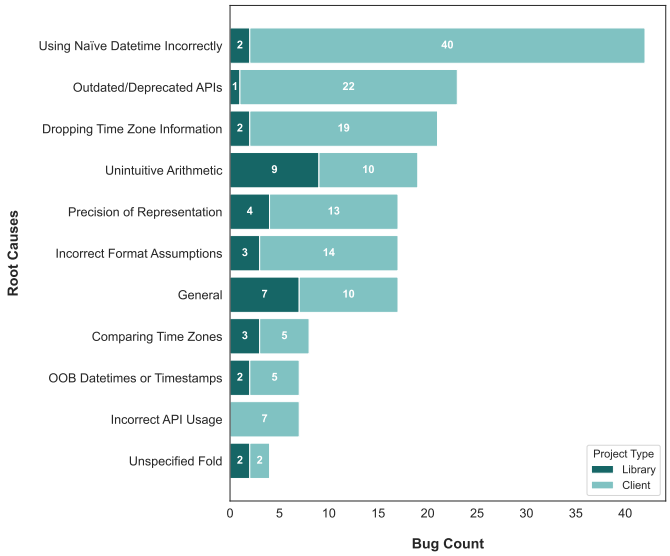


Fig. 5: Distribution of bugs across the identified root causes. Incorrect use of naive `datetime` objects emerges as the largest category.

> 💡 **Insight #2:** Despite the inherent complexities of date and time arithmetic [16, 35], our analysis reveals that the most prevalent bugs arise from the incorrect construction of `datetime` objects. This underscores the essential need for accuracy in constructing `datetime` objects, as initial errors can compromise data integrity and undermine subsequent operations' reliability.

C. What kinds of errors lead to the date/time bugs?

Figure 5 shows the taxonomy we developed for the various root causes of date/time bugs after analyzing their fixes.

Using Naive `Datetime` Incorrectly (*27.8%*): In agronholm/apscheduler#133, a bug arises due to the mixing of naive and time-zone-aware `datetime` objects. Specifically, the task scheduler operates in UTC but the system time is in a different time zone. This causes local time to be incorrectly treated as UTC leading to massive errors in task scheduling, literally the main job of the entire application! Other examples, like pallets-eco/flask-session#85, include the use of naive `datetime` objects when the application should have been using time-zone-aware objects.

Outdated/Deprecated APIs (*15.2%*): Using deprecated APIs can cause problems in the future. For example, nextcord/nextcord#1062 moves away from using `datetime.utcnow()` and `datetime.utcfromtimestamp()` methods, which are deprecated in Python 3.12, and switch over to a time-zone-aware API.

Dropping Time Zone Information (*13.9%*): In googleapis/google-cloud-python#131, an issue arises where one of the APIs discards time zone information in its internal representation, while another API consistently uses time-zone-aware objects. This discrepancy leads to type errors during comparisons. Other examples of related issues include failing to write time zone information to a database or not parsing it from the provided format string when constructing `datetime` objects.

Unintuitive Arithmetic (*12.6%*): For example, in sdispater/pendulum#475 the 'duration.in_days()' methods in *Pendulum*, fails to account for all the components of the `datetime` object (years, months, seconds...) and ultimately produces incorrect results.

Precision of Representation (*11.3%*): In influxdata/influxdb-client-python#455, the code loses nanoseconds information due to the internal representation not being precise enough. On the other hand, in art049/odmantic#99, the creation of a model instance containing `datetime` fields fails if the microseconds are between 999500 and 999999 because the information is more precise than required!

Incorrect Format Assumptions (*11.3%*) A bug in scrapinghub/dateparser#615 involves an incorrect assumption on the format of input date/time strings; specifically, it assumes that the time zones offsets cannot contain double digits. The regex code involved in parsing silently fails! Similarly, pyopenapi/pyswagger#83 assumes ISO 8601 timestamps do not contain milliseconds leading to milliseconds as well as time zone information being dropped! dateutil/dateutil#292 occurs since the `strftime()` method fails when the input datatime objects contain time zone information as a `tzfile`.

Comparing Time Zones (*5.3%*): As discussed earlier, dateutil/dateutil#151 and pydantic/pydantic#8683 involved issues with equality-based comparisons across distinct Python objects representing the same logical time zone.

Incorrect API Usage (*4.6%*): For example, in agronholm/apscheduler#444 incorrectly limits the valid inputs for an interval to be integers; however, the underlying `timedelta` API accepts floats. In trinodb/trino-python-client#366, *Pytz* time zones are used incorrectly (`localize()` is not invoked on the `datetime` objects) resulting in the time zone being set as Local Mean Time (LMT) and the offset becoming *53 minutes*.

OOB `Datetimes` or Timestamps (*4.6%*): In jborean93/smbprotocol#114, an `OverflowError` can arise while trying to parse timestamps from file headers since they can contain values greater than the max `datetime` constant set in Python (`9999-12-31T23:59:59.999999Z`). Similarly, the issue in SEED-platform/seed#1439 occurs when the year 1888 is passed to `strftime()` which requires years to be at least 1900.

Unspecified Fold (*2.6%*): Date/time computations can return incorrect results due to the lack of information to disambiguate wall-clock values during a DST transition where the clock rolls back. For example, in mverleg/pyjson_tricks#89 the bug arises because there is no `fold` attribute associated with `datetime` objects and in sdispater/pendulum#767 the bug arises because the `deepcopy()` method does not account for the correctly populated `fold` attribute.

General (*11.3%*): For example, dateutil/dateutil#216 is a issue that occurred since the windows specific `tzwinlocal()` API failed to work with the *Dateutil* in cases where `tzwinlocal()` pointed to time zones observing DST. Another example is sdispater/pendulum#167 where the code fails in the Windows 10 bash shell due to an unhandled edge case of the time zone string being empty. Other types of general root causes include hard-coding return types or compatibility issues with different library versions.

> 💡 **Insight #3:** The root causes of date/time bugs are diverse, but often involve nuances about date/time library APIs—such as assumptions, default conventions, or edge-case behavior—that can only be understood by carefully analyzing documentation and relating the information to the application context in terms of program requirements, operational expectations, and internal design choices.

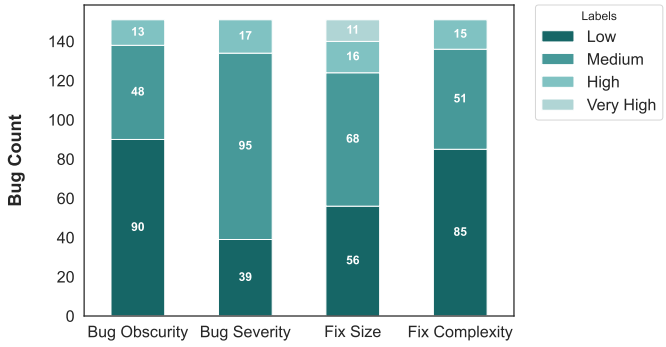
D. What is the nature of the bugs and their fixes?

Figure 6a captures the distribution of all the 151 bugs from our study along the different ordinal study factors. We describe each of these factors in detail.

Bug Obscurity: This term describes the level of difficulty involved in detecting or triggering a bug. *59.6%* of bugs exhibit a "*Low*" obscurity level, which we assigned to issues where the buggy behavior manifests upon the very first execution of the code, regardless of input. For example, googleapis/python-storage#1194 concerns the usage of outdated APIs that consistently raise a `DeprecationWarning` upon execution. Approximately *31.7%* of bugs fall under a "*Medium*" obscurity level, a label we assigned bugs that only manifest under certain inputs but when they do the failure is immediately apparent (e.g., due to an exception). The issue weewx/weewx#356 pertains to code that is affected by DST changes occurring at midnight, in which case the program fails with an explicit error message. Finally, *8.6%* of bugs are categorized under a "*High*" obscurity level; these represent edge-case bugs that can fail silently (e.g., by only producing a wrong output), making them challenging to detect. For instance, sdispater/pendulum#74 involves an off-by-one microsecond error that occurs silently and only in specific time zones.

Bug Severity: The severity of a bug is primarily determined by reviewing developer discussions and, if available, GitHub issue tags[2]. Approximately *25.8%* of bugs are classified as having "*Low*" severity, indicating that an error occurs during a non-critical task. Referring back to googleapis/python-storage#1194, deprecated API-related bugs typically only trigger warnings rather than causing errors or exceptions. About *62.9%* of bugs have a "*Medium*" severity level, suggesting that

[2]In case of insufficient data, we add a conservative label.



(a) Breakdown of bugs by obscurity, severity, localization, and fix complexity. Nearly *60%* of bugs demonstrate low obscurity, *∼74%* have a medium or high impact, *∼82%* are localized within a single function, and only *∼10%* require complex fixes.

(b) Correlation matrix illustrating relationships between ordinal study factors. The data shows a slight negative correlation between bug severity and obscurity (*-0.11*), and a minimal correlation between bug severity and fix size (*0.08*), suggesting that the potential impact of tooling for detecting/fixing shallow bugs should not be discounted.

Fig. 6: Distribution and analysis of 151 bugs across various ordinal study factors. Insights emphasize the efficacy of *Static Analysis* in identifying impactful bugs.

while some essential functionality is affected, workarounds are available. In stringertheory/traces#217, there's an issue when handling intervals containing `datetime` objects whose hour attribute is set; however, there are various ways to mitigate this error (e.g., normalizing all objects to remove the hour attribute). Lastly, *11.3%* of bugs are labeled as having "*High*" severity; these are assigned to bugs that disrupt critical functionality of the parent project, with the only remedy being a code fix. For example, in aws/aws-sdk-pandas#2410, the AWS S3 API fails to read an Apache Parquet object as a Pandas dataframe when records contain UTC-annotated timestamps.

The issue kiorky/croniter#56, in a library that allows using Cron-like job scheduling syntax in Python, is classified as both *high severity* and *high obscurity* since it produces the wrong schedule when run during a DST transition.

Fix Size: This factor assesses the size of the bug's related code fix in terms of the number of lines of Python code, serving as a proxy for the effort required to resolve the issue. Fix sizes are determined by manually reviewing pull requests to exclude potential code changes unrelated to the bug (e.g., updating test code). Each bug is categorized as follows: *37.1%* of bugs are annotated as "*Low*," indicating that the fix involved a single line of Python code or the same single-line syntactic change repeated multiple times. Approximately *45%* of bugs are labeled as "*Medium*," signifying multiple changes within the same Python function. Changes that affect multiple lines across various functions within the same Python file are designated as "*High*" (*10.6%*). Finally, bugs requiring extensive modifications across multiple Python files are labeled as "*Very High*" (*7.3%*). For example, in snowflakedb/snowflake-connector-python#926 the fix for an off-by-one microsecond bug required time-duration calculations across multiple files and hundreds of lines of code to be updated.

Fix Complexity: This metric evaluates the logical complexity required to modify code to correct a bug. Each bug is labeled by analyzing the specific code changes made in the fix. Approximately *56.3%* of the bugs are labeled as "*Low*", indicating that the fix is straightforward. In nautobot/-nautobot#2426, a job scheduling bug arises due to indexing mismatch on weekdays, i.e. `datetime` treats Monday as the first day of the week whereas `crontab` treats Sunday as the first day of the week. The fix involves a single-line change to the computed index. A "*Medium*" label is assigned to *33.8%* of the bugs, where the fix involves resolving a single logical error within the code. For example, Pycord-Development/pycord#2187 a task repeats a hundred billion times because the time zone is not set to UTC. The fix involves making use of consistent time zone information during initialization, requiring updates to a few conditional predicates. Finally, a "*High*" label is given to *9.9%* of the bugs, which require addressing multiple logical errors throughout the program. For issues like snowflakedb/snowflake-connector-python#926 (previously discussed), the fix involves redesigning methods, refactoring code, and updating arithmetic logic.

Correlations between Factors: Having done the labeling, we wondered if severe bugs were hard to detect and/or fix. Figure 6b presents the correlation plot for the ordinal factors in our study. Interestingly, we found that the severity of a bug and the difficulty of detecting it exhibit a slight negative correlation (*-0.11*). Additionally, the amount of code altered to rectify a bug is nearly uncorrelated (*0.08*) with the severity of the bug being addressed.

> 💡 **Insight #4:** Our analysis, as depicted in Figure 6a, indicates that a significant proportion (*∼60%*) of bugs are readily detectable due to low obscurity, while approximately *82%* of the fixes are confined within a single function's scope, suggesting a high degree of localization. Moreover, Figure 6b shows that severe bugs are not necessarily hard to detect and/or fix.

V. Discussion

What are the implications of our findings for software engineering practice and research? We next describe two ways

in which the insights uncovered from our empirical analysis can be used to improve software quality in the future.

A. Proactively Identifying Bugs via Domain-Specific Testing

Insight #4 suggests that a majority of the studied date/time bugs (∼60%) manifest readily when the erroneous code is executed with *any* input; so, in theory, they should be detectable during continuous integration via regular testing with comprehensive test suites that have sufficient code coverage and high-quality test oracles. Of course, the projects in our study lacked such tests, which is why the bugs went undetected in the first place. Moreover, another ∼32% of the bugs were classified as *medium obscurity*; that is, they can be detected immediately (e.g., via a crash) when the buggy code is executed with the right inputs. What, concretely, can developers do to proactively identify such issues?

We look to Insight #1, which suggests that a large proportion of the studied date/time bugs involve dealing with either time zones or string representations, as well as Insight #2, which suggests that most mistakes occur when constructing date/time values. So, we make the following three recommendations to harden software against common but shallow classes of date/time bugs.

1) **Maximize coverage of temporal characteristics with domain-specific test cases and environments**: To begin with, test suites should be comprehensive enough to include coverage of critical date/time computations. Moreover, for every date/time value that is derived either from the system clock or from an external input (such as a file, database, or web form) the test suite should include cases that vary characteristics such as calendar date and time zone. System-clock-related operations can be mocked using Python libraries like *Freezegun* [36] and OS-level utilities such as *Libfaketime* [37]. Many bugs in our dataset could have been identified if the corresponding code was simply executed and validated from a particular time zone.
2) **Harden parsing and serialization logic via property-based testing (PBT)**: For code that converts between date/time values and strings via files or databases, property-based testing using libraries such as *Hypothesis* [38] may be effective at validating *round-trip* invariants—for example, that a sequence of first serializing then parsing (or vice versa) date/time values retains the original value. Several bugs in our dataset could have become evident if such a round-trip property had been checked for any input.
3) **Use random fuzz testing for discovering edge-case bugs:** Many of the bugs related to out-of-bounds values (e.g., years before 1900), precision issues (e.g., fractional microseconds), and several types of time-zone and DST-related issues could potentially be detected via a simple form of fuzz testing—simply executing code with randomly sampled input values. The random input generation can be achieved with PBT tools such as *Hypothesis*, but for these types of bugs no special oracle is needed; simply checking for the absence of run-time exceptions may be sufficient to catch the low-hanging fruit.

We note that the high-obscurity bugs (i.e., those that fail silently) comprise less than 10% of the overall bugs we studied; moreover, as per Insight #1, less than 8% of the bugs were related to DST. While it is possible that these types of bugs indeed occur rarely, it is also possible that such bugs are simply under-reported because of the fact that they are hard to detect and manifest infrequently in production. In fact, such bugs may be lying dormant in current projects. We next turn to ways in which such bugs could possibly be uncovered using static program analysis.

B. Uncovering Dormant Bugs via Static Analysis

Insight #4 indicates that many bugs can be traced to just one or a few lines of code within a single function. We speculate that static analysis tools could be useful for identifying common classes of dormant date/time bugs through syntax-based pattern matching. To explore these implications, we conducted a prototype evaluation using an off-the-shelf pattern-based static analysis tool to find specific categories of date/time bugs.

Case Study—Using CodeQL to detect `datetime` construction bugs: We employ the CodeQL [39] tool to identify date/time construction errors that may lead to computations with buggy edge-case behavior: (a) *Deprecated API Usage:* flags the use of outdated methods like `datetime.utcnow()` and `datetime.utcfromtimestamp()`, which return naive (i.e., time-zone-unaware) objects representing UTC time—these are dangerous because other Python library functions treat any naive timestamps as representing the user's local time (which may be different from UTC); (b) *Custom Fixed Time Zones:* detects the use of custom fixed-offset time zones that ignore DST changes (e.g., "UTC-04:00" instead of "US Eastern Time") which if combined with duration arithmetic can produce incorrect results; and (c) *Partially Replacing Components:* identifies operations where objects are improperly constructed by partially modifying individual date components (such as year, month, or day)—these operations can fail if the result is a non-existent value.

We ran these static analyses on 1,000 randomly selected GitHub repositories from our dataset, identifying several dormant bugs. Our CodeQL query for deprecated API usage raised 617 alerts—these should all be true positives, as deprecated APIs are inherently problematic[3]. The fixed-offset time zone query identified 23 results, with only 1 being an actual bug which we reported[4]; the rest were false positives which we manually identified by observing the context. For example, in intel-extension-for-transformers a custom fixed-offset time zone of UTC+8 was used specifically for Shanghai

[3]Filed issues GoogleCloudPlatform/cortex-data-foundation#78 and tylere-bowers/Schwabdev#23.

[4]Filed issue XzwHan/CARD#24.

(indicated by a string literal); this is not a bug because China does not observe DST changes. The query for incorrect partial replacement returned 37 results, of which only 2 were true positives which we reported[5]. False positives were again identified by manual analysis. For example, one of the true bugs we found in stocks prediction module involved computing a date exactly two years ago from today via the expression "`now().replace(year=now().year-2)`"; this is a bug since it can fail when executed on a leap day (Feb 29)! In contrast, our analysis flagged a similar `replace` operation in clangen, but this was not a bug: "`date.today().replace(year=2000)`" always works because the year 2000 is a leap year.

Implications: Pattern-based static analysis can identify code smells, but distinguishing true bugs from false positives requires analyzing additional context from related software artifacts. This aligns with our Insight #3, highlighting that context is crucial for identifying bugs. Further research is necessary to develop advanced techniques that can incorporate contextual information to accurately detect true bugs.

VI. Related Work

To the best of our knowledge, this is the first academic effort to systematically study software engineering bugs related to date/time computations.

Grey Literature: The difficulties of getting temporal computations right are known to developers. There are articles documenting the incorrect assumptions programmers often make about date/time-related computations [8, 9]. There also exist articles that describe the nuances of different Python date/time libraries and their semantics [10, 35, 40]. Developers often write about specific problems they faced and their learnings [13, 41, 42, 43]. We believe that this body of literature serves as a motivating factor for our comprehensive and systematic study, which aims to illuminate the landscape of date/time-related bugs and identify strategies for enhancing the reliability of software systems.

Academic Research: Despite the existence of considerable grey literature, date/time bugs have received relatively little attention in academic research. Some prior work focused on detecting well-defined date-related bugs, such as the Y2K problem with string formats [44] and the Y2038 problem related to 32-bit Unix timestamps [45, 46]. Database researchers have studied various aspects of storing and querying temporal values [47, 48, 49, 50, 51]. Recently, Monat et al. [52] formalized semantics for date and duration arithmetic and demonstrated its application to verifying legal texts. We are not aware of any tooling for symbolically reasoning about arbitrary date/time computations in general-purpose languages like Python.

Empirical Studies and Bug Detection Tools: Numerous studies on software bugs have proven invaluable for enhancing the reliability of software systems. Researchers have investigated the causes of bugs in cloud systems [53, 54], infrastructure as code environments [28], and open-source projects [55]. Insights gained from these studies have often led to the creation of innovative bug-detection tools that significantly improve system reliability [56]. Drawing inspiration from these empirical studies, we aim to contribute to strengthening the correctness of the date and time computations in software.

[5]Filed issues Leci37/TensorFlow-stocks-prediction-Machine-learning-RealTime#34 and 13812851221/-rxrw-daily_morning#31.

VII. Dataset Availability

Our fully annotated dataset and analysis scripts are available at https://github.com/cmu-pasta/date-time, containing: (1) scripts for extracting and ranking likely date/time bugs (Section III), (2) the curated dataset of 151 bugs with fix links and annotations across 7 study factors (Section IV), and (3) CodeQL queries and scripts used to run them on 1,000 random repositories (Section V).

VIII. Conclusion

We conducted the first in-depth systematic study of date/-time bugs in open-source software. Our qualitative analysis on 151 bugs from Python-based GitHub projects revealed several key insights: most bugs (*53.6%*) are related to time zones, with incorrect construction of `datetime` objects being the primary source of error (*58.9%*). Additionally, many errors stem from misconceptions about library API behavior. Lastly, most bugs (*∼82%*) are confined to a single function and are easily fixable; based on this finding, we hypothesize that static analysis tools could effectively detect date/time bugs and that such bugs can potentially be fixed automatically. We also made concrete recommendations to developers for strengthening their software against temporal computation bugs via automated testing strategies; for example, simply increasing test coverage and having appropriate oracles could have revealed *∼60%* of the bugs we studied, whereas time-zone-mocking or property-based/fuzz testing can potentially uncover other input-dependent bugs as well. Our publicly available dataset aims to benefit researchers and practitioners, contributing to more reliable software systems and inspiring new research directions.

Acknowledgments

This work was supported in part by the National Science Foundation via grants OAC-2244348, CCF-2120955, and CCF-2429384.

References

[1] Investopedia, "IRS error hits some disaster-affected California taxpayers with incorrect due dates," https://www.investopedia.com/irs-error-hits-some-taxpayers-with-incorrect-due-dates-7509720, 2023, retrieved March 19, 2024.

[2] KFF Health News, "Like clockwork: How daylight saving time stumps hospital record-keeping," https://kffhealthnews.org/news/like-clockwork-how-daylight-saving-time-stumps-hospital-record-keeping/, 2018, retrieved March 19, 2024.

[3] Formula 1, "Software glitch cost Hamilton victory – Mercedes," https://www.formula1.com/en/latest/article/software-glitch-cost-hamilton-victory-mercedes.6VzyCYpEpauaIYsOWYCqYS, March 2018, retrieved March 16, 2024.

[4] C. Neal, "Old certificate, new signature: Open-source tools forge signature timestamps on Windows drivers," https://blog.talosintelligence.com/old-certificate-new-signature/, July 2023, retrieved March 19, 2024.

[5] Reuters, "'leap year glitch' shuts some New Zealand fuel pumps," https://www.reuters.com/world/asia-pacific/leap-year-glitch-shuts-some-new-zealand-fuel-pumps-2024-02-29/, February 2024.

[6] D. Cheian, "I see dead patents: How bugs in the patent system keep expired patents alive," *Fordham Intell. Prop. Media & Ent. LJ*, vol. 33, p. 1, 2022.

[7] Daily Tech, "Lockheed's F-22 raptor gets zapped by international date line," http://www.dailytech.com/article.aspx?newsid=6225, February 2007, archived on March 16, 2007 by web.archive.org.

[8] N. Sussman, "Falsehoods programmers believe about time," https://infiniteundo.com/post/25326999628/falsehoods-programmers-believe-about-time, retrieved October 29, 2024.

[9] ——, "More falsehoods programmers believe about time; "wisdom of the crowd" edition," https://infiniteundo.com/post/25509354022/more-falsehoods-programmers-believe-about-time, retrieved October 29, 2024.

[10] A. Bovenberg, "Ten Python datetime pitfalls, and what libraries are (not) doing about it," https://dev.arie.bovenberg.net/blog/python-datetime-pitfalls/#imagining-a-solution, January 2024, retrieved October 29, 2024.

[11] OpenAI, "GPT-4: Technical report," *arXiv preprint arXiv:2303.08774*, 2022. [Online]. Available: https://arxiv.org/abs/2303.08774

[12] Python, "datetime — basic date and time types," https://docs.python.org/3/library/datetime.html, retrieved November 2, 2024.

[13] T. Mick, "The problem with Python's datetime class." https://lo.calho.st/posts/python-datetime-problems/, retrieved November 2, 2024.

[14] S. Eustace, "Pendulum: Python DateTimes made easy," https://pendulum.eustace.io/docs, retrieved March 19, 2024.

[15] Arrow Collective, "Arrow: Better dates & times for Python," https://arrow.readthedocs.io, retrieved March 19, 2024.

[16] A. Bovenberg, "Whenever: Sensible and typesafe datetimes," https://whenever.readthedocs.io, retrieved March 19, 2024.

[17] G. Niemeyer, T. Pieviläinen, Y. de Leeuw, and P. Ganssle, "dateutil - powerful extensions to datetime," https://dateutil.readthedocs.io/en/stable/, retrieved March 19, 2024.

[18] S. Bishop, "pytz," https://pypi.org/project/pytz/, retrieved November 2, 2024.

[19] P. Nilsson, "Heliclockter," https://pypi.org/project/heliclockter/, retrieved November 2, 2024.

[20] K. Reitz, "Maya: Datetimes for humans," https://github.com/kennethreitz/maya, retrieved November 2, 2024.

[21] M. Yusuf, "Delorean: Time travel made easy," https://delorean.readthedocs.io/en/latest/, retrieved November 2, 2024.

[22] Z. Williams, "moment," https://github.com/zachwill/moment, retrieved November 2, 2024.

[23] F. Wiegand, "Chronyk," https://github.com/KoffeinFlummi/Chronyk, retrieved November 2, 2024.

[24] GitHub, "GitHub GraphQL API documentation," https://docs.github.com/en/graphql, retrieved October 30, 2024.

[25] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[26] A. Strauss and J. M. Corbin, *Grounded Theory in Practice*. Sage, 1997.

[27] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[28] G.-P. Drosos, T. Sotiropoulos, G. Alexopoulos, D. Mitropoulos, and Z. Su, "When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689799

[29] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents? an empirical study on a large-scale cloud service," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 126–141. [Online]. Available: https://doi.org/10.1145/3542929.3563482

[30] A. Eghbali and M. Pradel, "No strings attached: an empirical study of string-related software bugs," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 956–967. [Online]. Available: https://doi.org/10.1145/3324884.3416576

[31] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 155–162. [Online]. Available: https://doi.org/10.1145/3317550.3321438

[32] TIOBE, "TIOBE Index," https://www.tiobe.com/tiobe-index/, retrieved November 9, 2024.

[33] L. Regebro, "PEP 431 – Time zone support improvements," https://peps.python.org/pep-0431/, retrieved November 2, 2024.

[34] A. Belopolsky and T. Peters, "PEP 495 – Local Time Disambiguation," https://peps.python.org/pep-0495/, retrieved November 2, 2024.

[35] P. Ganssle, "Semantics of timezone-aware datetime arithmetic," https://blog.ganssle.io/articles/2018/02/aware-datetime-arithmetic.html, June 2020, retrieved October 29, 2024.

[36] S. Pulec, "FreezeGun: Let your Python tests travel through time," https://github.com/spulec/freezegun, retrieved November 2, 2024.

[37] W. Hommel, "libfaketime," https://github.com/wolfcw/libfaketime, retrieved November 5, 2024.

[38] D. R. MacIver, "Hypothesis," https://hypothesis.readthedocs.io/en/latest/, retrieved November 2, 2024.

[39] GitHub, "CodeQL," https://codeql.github.com/, retrieved November 5, 2024.

[40] P. Ganssle, "A curious case of non-transitive datetime comparison," https://blog.ganssle.io/articles/2018/02/a-curious-case-datetimes.html, October 2021, retrieved October 29, 2024.

[41] L. Taarnskov, "How to save datetimes for future events - (when UTC is not the right answer)," https://www.creativedeletion.com/2015/03/19/persisting_future_datetimes.html, March 2015, retrieved October 29, 2024.

[42] P. Ganssle, "pytz: The fastest footgun in the west)," https://blog.ganssle.io/articles/2018/03/pytz-fastest-footgun.html, April 2020, retrieved November 5, 2024.

[43] ——, "Why naïve times are local times in Python," https://blog.ganssle.io/articles/2022/04/naive-local-datetimes.html, April 2022, retrieved November 5, 2024.

[44] M. Elsman, J. S. Foster, and A. Aiken, "Carillon—a system to find Y2K problems in C programs," *Technical Report*, 1999.

[45] K. Suzuki, T. Kubota, and K. Kono, "Detecting and analyzing year 2038 problem bugs in user-level applications," in *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2019, pp. 65–6509.

[46] R. Okabe, J. Yabuki, and M. Toyama, "Avoiding year 2038 problem on 32-bit linux by rewinding time on clock synchronization," in *2020 25th IEEE international conference on emerging technologies and factory automation (ETFA)*, vol. 1. IEEE, 2020, pp. 1019–1022.

[47] R. Snodgrass *et al.*, "Temporal databases," *Computer*, vol. 19, no. 09, pp. 35–42, 1986.

[48] R. Chandra, A. Segev, and M. Stonebraker, "Implementing calendars and temporal rules in next generation databases," in *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. IEEE, 1994, pp. 264–273.

[49] O. Etzion, *Temporal databases: research and practice*. Springer Science & Business Media, 1998, vol. 1399.

[50] T. Johnston and R. Weis, *Managing time in relational databases: how to design, update and query temporal data*. Morgan Kaufmann, 2010.

[51] C. Bettini, S. Jajodia, and S. Wang, *Time granularities in databases, data mining, and temporal reasoning*. Springer Science & Business Media, 2013.

[52] R. Monat, A. Fromherz, and D. Merigoux, "Formalizing date arithmetic and statically detecting ambiguities for the law," in *ESOP 2024 - 33rd European Symposium on Programming*, 2024, pp. 421–450.

[53] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 155–162. [Online]. Available: https://doi.org/10.1145/3317550.3321438

[54] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents? an empirical study on a large-scale cloud service," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 126–141. [Online]. Available: https://doi.org/10.1145/3542929.3563482

[55] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140. [Online]. Available: https://doi.org/10.1145/3213846.3213866

[56] B. A. Stoica, U. Sethi, Y. Su, C. Zhou, S. Lu, J. Mace, M. Musuvathi, and S. Nath, "If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-Informed Tooling for Detecting Retry Bugs in Software Systems," in *Proceedings of the ACM SOSP 2024*, 2024.