Understanding Commercial Low-code App Bugs

Hadia Syed*, Zhixing He*, Ye Tang*, Jiqing Liu*, Shengjie Chen§, Lixin Sun§ and Hao Zhong*†
*School of Computer Science, Shanghai Jiao Tong University, China
{hadiasyed, chandlr, tangye_22, vincent_ljq, zhonghao}@sjtu.edu.cn
§Inspur, China
{chenshj, sunlixin}@inspur.com

***Abstract*—As a paradigm of end-user programming, low-code development has attracted attention from academics and enterprises. Consulting firms estimate that this paradigm can have great business value and that many large enterprises can switch to low-code development. To understand the emerging paradigm, researchers conducted various empirical studies about low-code development. Still, most studies analyze bugs in low-code development platforms. Although these studies are related, low-code development platforms are classical programs, not low-code applications. Several other studies analyze how programmers perceive low-code applications, but do not provide direct analysis. As low-code applications are typically commercial, most low-code projects are not open source. It is difficult to collect the subjects, and many questions are still open.**

In this paper, we propose the first study on bugs in low-code applications. To overcome the above-mentioned challenge, we collect 8 actively maintained low-code projects from our industrial partner. From the 8 projects, we collected 248 bugs from our industrial partner and analyzed their symptoms, causes, and repairs. Our study derives 7 interesting findings. For instance, we find that 78.23% of low-code bugs are related to human-computer interactions. The result indicates that GUI testing can be critical in detecting bugs in low-code applications. As another example, we find that even if programs are developed on low-code platforms, modifying source files is necessary to fix 78.23% of low-code bugs.

***Index Terms*—Low-code application bug and empirical study**

I. Introduction

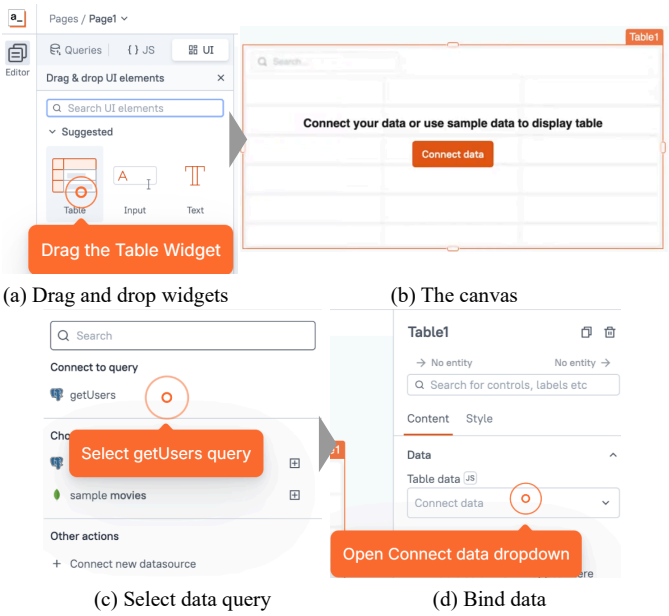
The demand for software systems increases rapidly, but software employees have not been growing at a sufficient pace [1]. As a result, it is difficult for many companies to hire sufficient professional programmers [2]. To reduce the requirement for programming expertise, end-user programming [3] has long been a hot topic. As a paradigm of end-user programming, some early discussions about low-code development can be traced to 2014 [4]. Compared with the classical paradigm, the low-code paradigm prefers low-code alternatives for fast, continuous, and test-and-learn delivery. Low-code development attracts much attention from researchers and practitioners. Consulting firms release positive reports about low-code development. For instance, a Forrester report [5] estimates that the market of low-code development will reach $21 billion by 2022, and a Gartner report [6] estimates that 65% of large enterprises will switch to low-code development to some extent by 2024. In 2025, Mendix surveyed more than 2,000 technical leaders, and their results [7] show that 98% of enterprises now use low-code platforms and tools in their development processes.

Researchers conducted various empirical studies on this emerging paradigm, and their studies can be roughly divided into two research lines. The first line of studies [8], [9] analyzes the characteristics of low-code development platforms (LCDPs). LCDPs provide IDEs, frameworks, and runtimes for low-code applications (LCAs). Although LCDPs support the development of LCAs, they themselves are not LCAs. Their findings do not show the characteristics of LCAs. The other line of studies [10], [11], [12] analyze Stack Overflow posts that discuss LCAs and LCDPs. Although their findings include opinions on LCAs, they are indirect and do not include the characteristics of LCA bugs. As most LCAs are developed in large enterprises and have significant business value, there are barely any LCAs in open-source communities, *e.g.*, GitHub. This is a major barrier to understanding the bugs in LCAs. As a result, no prior study has ever analyzed LCA bugs.

In this paper, we present the first empirical study on LCA bugs. To overcome the limitation of prior studies, we collect 8 commercial LCAs through our enterprise partner, and all 8 applications are under active maintenance. From these applications, we collected 248 bugs and manually analyzed their symptoms, causes, repairs, and associations. Our study explores the following research questions:

- **RQ1. What are the symptoms?**
  Motivation. The symptoms are useful for understanding the impact of LCA bugs on users.
  Protocol. We read bug reports and commit messages to build the taxonomy of symptoms.
  Answer. 78.23% of LCA bugs have symptoms in human-computer interactions (Finding 1). For the internal LCA bugs, 11.69% and 5.24% of bugs are about functionalities and wrong data, respectively (Finding 2).
- **RQ2. What are the causes?**
  Motivation. The results are useful for improving the quality of LCAs and detecting LCA bugs.
  Protocol. We read bug reports and buggy source files to build the taxonomy of causes.
  Answer. 70.56% of LCA bugs are caused by the lack of project-specific knowledge (Finding 3), and the remaining bugs (29.44%) are introduced due to the lack of domain-specific knowledge (Finding 4).
- **RQ3. What are the repairs?**
  Motivation. The results are useful for understanding how

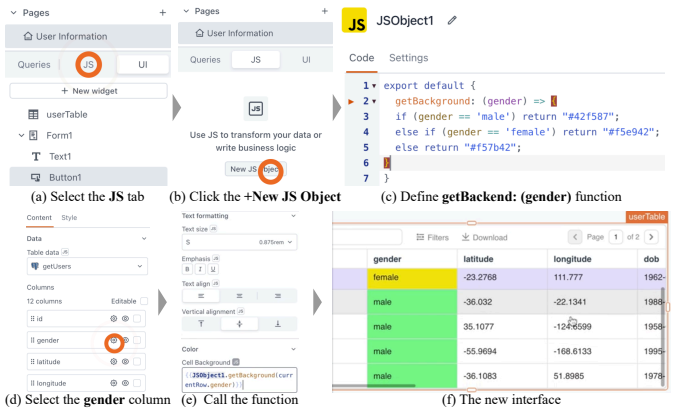
† Corresponding author: Hao Zhong (email: zhonghao@sjtu.edu.cn).



(a) Drag and drop widgets (b) The canvas

(c) Select data query (d) Bind data

Fig. 1. Reusable components of LCDP.



(a) Select the **JS** tab (b) Click the **+New JS Object** (c) Define **getBackend: (gender)** function

(d) Select the **gender** column (e) Call the function (f) The new interface

Fig. 2. An example of human-written code in LCDP.

to fix bugs in LCAs.

Protocol. We read bug reports and patches to understand the taxonomy of repairs.

Answer. In 88.31% of LCA bugs, modifying source code is essential (Finding 5). For the remaining bugs, changing enable conditions and modifying GUI settings repair 8.06% and 3.62% of LCA bugs, respectively (Finding 6).

- **RQ4. What are the correlations between symptoms, causes, and repairs?**

  Motivation. The results are useful for diagnosing the symptoms, causes, and repairs of LCA bugs.

  Protocol. We calculate the distributions of symptoms, causes, and repairs. We then calculate their correlations.

  Answer. Finding 7 shows that most causes have strong correlations with specific repairs.

Section II motivates our study, and Section V interprets the significance of our findings.

II. Motivating Example

This section introduces the differences between LCDPs and classical IDEs (Section II-A), a LCA bug (Section II-B), and the comparison with prior studies (Section II-C).

A. LCDP

LCDPs provide more reusable components than classical IDEs [13]. With their support, even non-professionals can quickly build the sketch of an application. For instance, Appsmith [14] is a popular LCDP, and Figure 1 shows the workflow of building a simple application with Appsmith. This application has a table to present some data. As shown in Figure 1(a), a programmer can drag a table and drop it to the application. By clicking this table, as shown in Figures 1(c) and (d), the programmer can bind a database connection to this table. After the programmer selects a suitable connection, this application can automatically retrieve the data for the table.

Although LCDPs reduce the effort of programming, they allow programmers to write code to implement customized logic. For instance, Figure 2 is a simplified tutorial of Appsmith [15]. As shown in Figure 2(a) and (b), a programmer can click the "JS" tab to implement the functionality of "New JS Object". Figure 2(c) shows the implemented method called `getBackground()`. This method returns a specific hexadecimal color based on the given gender. As shown in Figure 2(d) and (e) the programmer can select the table and click the setting of the background color property. In this way, the developer can add a short JavaScript code to invoke the `getBackground()` method. As shown in Figure 2(f), the gender cells of the built table have different background colours according to their contents.

B. LCA Bug

Although LCDP reduces the effort of implementing LCAs, LCAs can still have bugs. For instance, ***sfndpr is an LCA of water pricing systems. It supports submitting and publishing price policies. Figure 3 shows a bug in this LCA. In this table, each row presents a price policy. In particular, the first row lists a pricing policy for low-income families. The columns of this table list IDs and other details. The buttons above the table implement various functionalities like adding, deleting, and modifying the policy. In particular, a junior user can submit a price policy for approval by clicking the "Submit for approval" button. Before a policy is published, a senior user must approve it. Before senior users approve a policy, this policy shall not be published. However, due to this bug, in the second row, a policy is published before it is approved.

To manage the status of a price policy, programmers implement an automaton as follows:

```
1 publishPolicy () { ...
2   if ( uiState ['ids'].length != 0) {...
3   if ( uiState ['ids'].length != 0){
4     action$ = this.beActionService.invokeAction('policystatechange', ....) ;
5     ids = uiState ['ids'];
6     ... // return }}
```

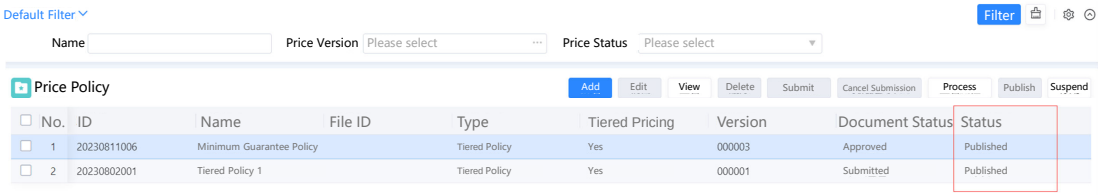


Fig. 3. A wrong procedure bug.

The `publishPolicy` function in the above code snippet handles the action of clicking the "Publish" button. In particular, Line 2 of this method determines whether the "Publish" button should be enabled. This line only checks whether the state length is zero, but does not check the state of the policy. As a result, a policy can be published before it is approved.

To fix this bug, the programmer adjusts the state transitions of the above automaton as follows:

```
1  publishPolicy () { ...
2      let ids = [];
3      if ( uiState ['ids'].length != 0){
4  -       ids = uiState['ids'];
5  +       uiState['rows'].forEach((row, idx, array) =>{
6  +           if(row['billstate'].billState == 'Approved'){
7  +               ids.push(row['id']);
8  +           }
9  +       })
10         const body = {
11             ids: ids,
12             ...
13         }
14         const action$ = this.beActionService.invokeAction('
               policystatechange', ..., body);
15     ... // return
16 }
```

In the code above, `publishPolicy` is responsible for determining which policies can transition to the "Published" state. The original implementation incorrectly assigned all policy IDs from `uiState['ids']` without checking their approval status, allowing unapproved policies to be published. In the fix, the programmer adds logic to filter the policies based on their state. Specifically, the new code iterates over `uiState['rows']` and only includes policies with the billState set to "Approved". This ensures that only approved policies are collected in the `ids` array, which is subsequently used for publishing. Consequently, the corrected implementation enforces the expected state progression, preventing the direct publication of unapproved policies.

As shown in Sections II-A, programmers write much fewer lines of code when implementing LCAs. As a result, many bugs in classical applications may not appear in LCAs. Even if some bugs appear in LCAs, their shapes can be different from classical applications as shown in Section II-B. Although researchers have built various bug taxonomies (See Section VI for details), it is interesting to build a taxonomy for the new programming paradigm of LCAs.

C. Comparison

Only three studies [10], [11], [12] analyze LCSDs, and none of them analyzes LCA bugs directly. Our study is novel in terms of its subjects, analysis methodology, and findings:

1. Subjects. The prior studies do not analyze LCA bugs but the posts from Stack Overflow [10], [11], [12] and Reddit [10]. As a comparison, we analyze real-world LCA bugs that are collected from a reputable enterprise. While their subjects convey personal experience and second-hand knowledge, our real-world LCA bugs make our study more reliable.

2. Protocols. The prior studies analyze Stack Overflow and Reddit posts, but LCA bugs and their programming contexts are seldom mentioned in such posts. In contrast, when analyzing LCA bugs, we have full access to all details. For instance, in Section II-B, we can analyze the symptoms, the causes, and the repairs of LCA bugs. In addition, when they analyze posts, it is difficult to obtain feedback from the authors, but we can ask LCA programmers for feedback.

3. Mutually corroborating findings. Despite the different subjects and protocols, we have some mutually corroborating findings. For instance, Luo *et al.* [10] report that the top languages in LCAs are Java and JavaScript. Table I shows that our LCA subjects have many Java and script source files. As another example, Luo *et al.* [10] report that APIs and templates are the most interesting domain knowledge in developing LCAs. Our identified ID mismatch bugs can be caused by the inconsistencies between API calls and template definitions. Alamin *et al.* [11], [12] report that 30% of posts discuss customization. We find that 11.69% of LCA bugs are fixed by modifying customizations.

4. Different and new findings. Some findings are relevant but different. For instance, many programmers believe that non-technical people can create LCAs with LCDPs [10]. As shown in Table I, we confirm that several programmers can implement large-scale LCAs, but repairing LCA bugs still requires much programming experience since modifying source files is essential in repairing 88.31% of LCA bugs (Finding 5). Besides the different findings, we derive some new findings. For instance, the prior studies [11], [12] build taxonomies for LCSDs, but we build taxonomies for LCA bugs. As a new programming paradigm, LCA bugs are different from those bugs in deep learning libraries [16], machine learning systems [17], and blockchain systems [18]. We complement these studies with the taxonomies for a new type of application.

Researchers have conducted studies to analyze bugs in other software systems. Section VI provides a brief survey on such studies. Although we do not find unique LCA bugs, we find that their distributions are different. For instance, Zhang *et al.* [19] and Jia *et al.* [20] report that libraries do not have GUI bugs. Akond and Farhana [21] report that 38.2% of bugs

are related to GUIs in covid19 software projects. Zhenmin *et al.* [22] report that 52.7% of bugs are related to GUIs in Mozilla. As a comparison, we find that 78.2% of LCA bugs are related to human-computer interactions.

III. Methodology

In this section, we introduce our analysis methodology.

A. Dataset

Table I shows our dataset. Column "Application" lists the names of the LCAs. In particular, ***mbp and ***mcf are water billing systems. ***mpb is a water debt system. ***mim is a receipt management system. ***mom and ***orderes are order systems. ***rmr is a meter reading system. ***sfndpr is a water pricing system. ***odm is a document management system. Column "LOC" denotes the lines of code for each project. Sub-columns "Java", "Script", "Markup", and "Format" denote the lines of code in Java, type languages, markup languages, and data formats (*e.g.*, json), respectively. In total, the 8 projects have more than 500,000 lines of code, and most code lines are implemented in script languages, *e.g.*, JavaScript. Column "Prog" shows the number of programmers. Due to the benefits of low-code development, most projects are implemented by fewer than ten programmers. Column "Start" denotes the time of submitting the first commit. As low-code development is an emerging paradigm, most projects have been developed since 2023. Column "Bug" shows the number of our inspected bugs. As most projects are young, we do not collect many bugs. In addition, it is expensive to analyze bugs manually. Still, the size of the subjects is comparable with the prior studies. For instance, Romano *et al.* [23] and Wu *et al.* [24] analyze 146 and 347 compiler bugs, respectively.

From industrial partners, we collect eight LCAs from a world-leading low-code software enterprise. These LCAs are implemented for government digitalization and enterprise digital transformation, *e.g.*, water fee pricing systems. Besides developing LCAs, this enterprise implements its own LCPD including the IDE, the runtime, and the frameworks. Most frameworks and the IDE have already been released to a well-known open-source community. However, our industrial partner asks us not to release their company name. As a result, we anonymize the enterprise and the project names. All the projects are under active maintenance. Because our LCAs have business value, we cannot release their source code within a replicate package. Still, we are allowed to present code snippets so that readers can understand our results.

B. General Protocol

Like other companies, our surveyed company uses a `git` server to record revision histories. From this server, we extract all commits of the subject LCAs. We then search commit messages with keywords such as "bug" and "fault" for commits whose messages explicitly mention bugs. Besides the `git` server, the surveyed company has issue trackers to handle bug reports. For each bug fix, we search issue trackers to locate their bug reports. Most bug reports are obtained by matching bug IDs that are mentioned in commit messages. If a commit message does not mention a bug ID, we observe that a commit message and its corresponding issue report often share overlapping descriptions. Based on this observation, we calculate the Jaccard similarity coefficient [25] between commit messages and issue reports and select the best match. To ensure that we locate the correct bug reports, we manually read the modifications of commits and bug reports.

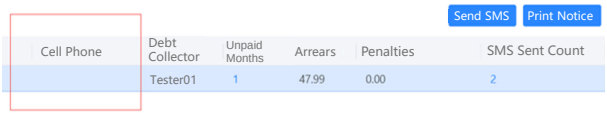


Fig. 4. A wrong display bug, where the cell phone is not displayed.

In RQs 1, 2, and 3, we explore the symptoms, causes, and repairs of bugs. Three authors independently read the commit messages and bug reports to build the taxonomy. After the initial taxonomy is built, the other two authors inspect their results. If there are any disagreements, we discuss them at our group meeting and make the final decision. If it is difficult to make the final decision, we consulted the corresponding application developers to clarify ambiguous cases. Besides the two sources, we analyze causes by reading the buggy source files. After the causes are determined, we read patches to understand the repairs. We adopted an open coding strategy [26] to systematically derive categories from the data. Through iterative coding and constant comparison, we summarized recurring patterns and merged similar concepts into higher-level categories. Based on the results of open coding and subsequent discussions with our industry partners, we built the taxonomy for each RQ. After we classified bugs, we asked our industry partners to review our classification results and revised them based on their feedback.

Researchers widely use Cohen's Kappa [27] to measure the inter-rater agreements. It produces a value that is between 0 and 1. In particular, a value closer to 0 indicates the least agreement, and 1 indicates the most agreement. Our calculated values were 0.97, 0.94, and 0.96 for symptoms, causes, and repairs, respectively. According to the values, we achieve almost perfect agreement on our built taxonomy.

IV. Result

This section presents our analysis results.

A. RQ1. Symptom

1) Protocol: In this research question, we classify the symptoms of LCA bugs. Although researchers [16], [17], [18] have built various taxonomies for bug symptoms, no prior study has built a taxonomy for LCA bugs. It is necessary to build the taxonomy for LCA bugs since LCA is implemented in a new programming paradigm as shown in Section II-A. As a result, we have to build our taxonomy. In this paper, we define the symptom of a bug as the unexpected behaviors that can be observed when bugs occur. After discussing with our partnering enterprise, we identify that most LCAs have GUIs

TABLE I
DATASET.

Application	Commit	LOC				Prog	Start	Bug
		Java	Script	Markup	Format			
***mbp	82	2,587	30,622	1,381	1,688	4	Aug/18/2023	12
***mpb	40	4,044	241,632	2,309	4,851	5	Apr/27/2023	13
***mcf	60	15,513	16,127	4,652	1,450	4	Sep/18/2023	24
***mim	28	2,084	18,264	2,996	1,598	4	Sep/20/2023	13
***mom	17	8,074	8,019	1,703	1,307	3	Sep/20/2023	4
***rmr	46	5,706	69,369	1,903	6,899	4	May/29/2023	7
***sfndpr	55	1,536	27,056	1,819	2,237	7	Apr/17/2023	10
***orderes	179	5,276	74,161	1,586	3,408	3	May/18/2023	6
***odm	2,087	12,598	10,389	6,303	13,617	24	Jul/5/2021	159
Total	2,594	57,418	495,639	24,652	37,055	58	-	248

and intensively interact with users. Based on this observation, we classify the symptoms of LCA bugs into two categories: human-computer interaction bugs and internal errors. For each bug, we analyze its bug report and fixing commit and classify them into one of the two categories. After that, we refine the bugs in each category into subcategories. In particular, we put each bug into an identified subcategory. If we cannot find a suitable subcategory, we build a new subcategory. Our taxonomy is incomplete, but it covers all analyzed bugs.

2) Results: We identify the following symptoms:

S1 Human-computer Interaction (194/248, 78.23%). In this category, bugs occur in the interactions between users and the software.

S1.1 Wrong Procedure (75/248, 30.24%). These bugs cause wrong procedures that are unaligned with the design of the software. For instance, the symptom in Figure 3 violates the designed procedure, so we put it in this category.

S1.2 Wrong Display (53/248, 21.37%). These bugs display data incorrectly. For instance, ***mpb is a water billing system. Figure 4 illustrates a bug in the user interface for requesting payments of water bills. In the table, each row presents a household, and the columns list IDs, names, addresses, mobile numbers, and other details. Above this table, users can send a billing message to the listed mobile number by clicking the "Send message" button. However, due to this bug, the first row fails to list the mobile number as highlighted in the red box.

S1.3 Wrong User Interface (39/248, 15.73%). In these bugs, programmers implement the wrong User interfaces. For instance, Figure 5 shows another bug in the water billing system, ***mpb. Above the table, users can send billing messages to customers by clicking the "Send SMS" button and print a notice by clicking the "Print Notice" button. If customers have already paid their bills, users shall not send any billing messages or print notices. In this case, both buttons must be disabled. However, due to this bug, as shown in the first row, when customers have already paid their bills, the two buttons are still enabled. The red rectangles highlight the two buttons. The total amount owed is zero, but the buttons are enabled. As a result, we put the bug into this category.

S1.4 Error Message (21/248, 8.47%). These bugs cause message boxes that display error details. For instance, ***mcf

Fig. 5. A wrong user interface bug, where the household in the red box has no arrear but can be sent a payment reminder SMS.

is a project for the water payment system. Figure 6 shows a bug. The GUI has three sections. The top section shows the details of the families, the middle section lists detailed information on arrears, and the bottom section lists payment details. Users can check the family information, choose a debt, and select the corresponding payment method. After that, the user can fill in the box labeled "Actual Received" with the actual received amount and press the "Pay" button to complete the payment. Due to this bug, a message box displays an error message when a user attempts to press the "Pay" button with an empty amount. Instead, the program should prompt a box and let users fill in the field. As a result, we put it in this category.

S1.5 Others (6/248, 2.42%). This category includes other types of human-computer interaction bugs. For instance, in ***odm, after users edit old documents in a certain format, they fail to print them. Printing bugs belong to special cases in the interaction between computers and users.

The above observations lead to a finding:

> **Finding 1.** Most LCA bugs (78.23%) are related to human-computer interactions.

GUI testing can be useful for detecting bugs in their human-computer interactions.

S2 Internal Error (54/248, 21.77%). In this category of bugs, programs produce internal errors. In most cases, programmers actively detect and repair these bugs.

S2.1 Functionality (29/248, 11.69%). In these bugs, programmers implement incorrect functionalities. For example, in ***odm, a programmer complains that messages cannot be sent if they are longer than 256 characters. As programmers

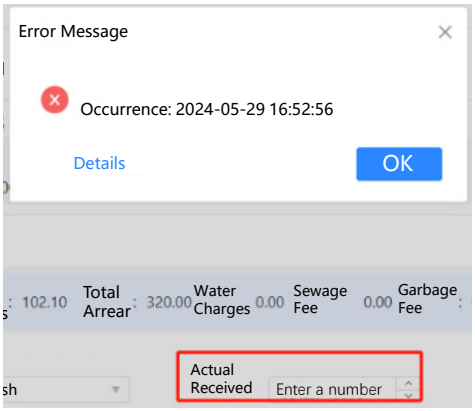


Fig. 6. An error message of a bug.

can bypass the bug, users may not notice its symptoms, but the internal functions do not work as expected.

S2.2 Wrong Data (13/248, 5.24%). If a bug causes the software to produce incorrect output data, we determine it as a wrong data bug. For example, ***mbp is a project that manages billing archives. When it imports a list of households, it provides non-sequential user IDs. As a result, some user IDs do not have corresponding households. Although users may not notice that their IDs are non-sequential, this bug wastes IDs, and we classify it as a wrong data bug.

S2.3 Others (12/248, 4.84%). This category encompasses various types of internal error bugs. For example, the LCA requires compiling the original design into a low-level language. In a specific version of ***odm, the project would encounter a bug during the compilation process and fail to compile. LCDPs use compilers to generate machine code, and their compilers can still have bugs. Before such bugs are fixed, LCAs must bypass the internal errors caused by compiler bugs.

The above observations lead to a finding:

> **Finding 2.** Most internal LCA errors are about functionalities (11.69%) and wrong data (5.24%).

Compared with crashes, it is more challenging to detect bugs about functionalities and wrong data, since it is difficult to obtain their test oracles.

In summary, most LCA bugs are GUI-related bugs. GUI testing can have some opportunities to detect LCA bugs. Most internal LCA bugs are about functionalities and wrong data. Different from crashes, there is no natural test oracle to detect such bugs. The research on test oracles can be critical in detecting LCA bugs.

B. RQ2. Cause

1) Protocol: This research aims to classify the causes of LCA bugs. Researchers [28], [29] have established taxonomies of causes when they analyze other types of software projects. Although they use the same term, researchers have different definitions for causes. Most researchers [30], [20] analyze bug causes from the perspective of source code. In our study, we analyze causes from the perspective of programmers. The low-code development paradigm is unfamiliar to programmers. Programmers introduce bugs because they do not have sufficient knowledge. From this angle, we identify two primary causes for LCA bugs after discussing with our partnering company. One category is the lack of project-specific knowledge, and the other category is the lack of domain-specific knowledge. After that, we further classify each category into specific subcategories.

2) Result: We classify causes into two categories.

C1 Project-specific Knowledge (175/248, 70.56%). Programmers introduce these bugs since they are unfamiliar with the knowledge specific to a project.

C1.1 Incorrect Logic (94/248, 37.90%). These bugs occur when programmers implement the wrong algorithms and logics. For instance, in ***odm, a bug report complains that a table of administrative organizations is unordered. This table is defined in the `DepartmentGroupMobile.hpl`:

```
{ ...
  "Content" : {
    "code" : "DepartmentGroupMobile",
    "sorts" : null, ...
    }
}
```

The above file defines business objects or application functionalities. Programmers can fill in the `sorts` field to select a sort method. This table is unsorted since programmers leave `sorts` as `null`. As the logic is wrong, we put this bug in the incorrect logic category.

C1.2 Data Retrieval (34/248, 13.71%). LCAs have many filters to retrieve data from databases. These bugs arise when programmers wrongly retrieve data from databases. For instance, in ***mpb, the `QueryDataVOAction` class declares the `getBqlCondition()` method. This method builds a query string, `bqlCondition`, to retrieve delinquent householders from a dataset. The patch for this bug is as follows:

```
1  private String getBqlCondition() {
2   String bqlCondition = "";
3   EntityFilter filter = getQueryContext().getFilter();
4   if (filter == null || filter.getFilterConditions() == null)
5       return "";
6   for (int i = 0; i < filter.getFilterConditions().size(); i++) {...
7       bqlCondition = bqlCondition + " and record.BusinessArea = '" +
8     filter.getFilterConditions().get(i).getValue() + "' ";...
9       }
10 +   //Number of Overdue Periods
11 +   bqlCondition = bqlCondition + " and case when recordCopy.times is
      null then 0 else recordCopy.times end <> 0";
12        ...
13      return bqlCondition;
14 }
```

In the above code, Lines 2 to 9 build a query string, and Line 13 returns the built string. LCA uses this string to retrieve delinquent householders, but it retrieves non-delinquent ones due to this bug. To repair this bug, Line 11 adds a comparison with the core delinquency status field, `recordCopy.times`. Here, this field records the number of delinquent bills for each householder. Line 11 adds a condition and it selects data whose delinquent bills are more than one. As programmers forget to add the comparison, the buggy code retrieves all householders

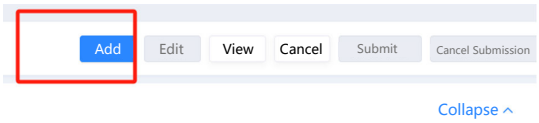


Fig. 7. A wrong user interface bug. The programmer misunderstood the requirement and did not disable the "Add" button in the red box.

including non-delinquent records. As this bug is caused by the retrieval of wrong data, we put this bug into this category and do not classify it to the incorrect logic category.

C1.3 GUI Design (32/248, 12.90%). These bugs are introduced because programmers fail to understand the correct UI designs. For instance, Figure 7 illustrates a bug in the water price system. In this bug, programmers fail to design when the buttons should be enabled. In particular, the system creates a new pricing policy after a user clicks the "Add" button. If the system works as designed, when a policy is created, users should not create another policy. As a result, during the process, the "Add" button should be disabled. However, as highlighted by the red rectangle in Figure 7, this button is enabled during policy creation. As a result, we determine that the GUI design is incorrect.

C1.4 Automata (11/248, 4.44%). When interacting with users, programmers often implement automata with various states. They can introduce bugs if the states are wrong. For instance, the bug in Figure 3 is caused by an unauthorized state transition, and we put it in the automata category.

C1.5 Others (4/248, 1.61%). This category includes other types of project-specific knowledge bugs, including incorrect time formats. For instance, in project ***mom, the programmer encounters a problem where the time filter does not work properly. The bug stems from the time filter not functioning correctly in the project. Specifically, in the `OrderManage.frm` file, the time filter was set with the following configuration:

```
1 "controltype" : "date",
2 "format" : "yyyy-MM-dd"
```

In the above code snippet, the filter only worked at a daily precision level, ignoring hours, minutes, and seconds. Consequently, the filter is unable to handle time-based precision beyond daily granularity. The time format setting belongs to special cases of project-specific knowledge.

The above observations lead to a finding:

> **Finding 3.** 70.56% of LCA bugs are introduced due to the lack of project-specific knowledge.

C2 Domain Knowledge (73/248, 29.44%). Programmers introduce these bugs since they have insufficient or incorrect domain knowledge of implementing low-code programs.

C2.1 ID Mismatch (40/248, 16.13%). To reduce the effort of manipulating data and calling across languages, low-code projects define many IDs to denote identifiers such as variable names, database columns, APIs, and web services. IDs are typically defined in the format of string values. For instance, Figure 4 shows a bug that fails to display mobile numbers. In the `DebtRecord.be.resource` resource file, the details of a user are associated with the following ID:

```
1 {
2     "id" : "***.DebtRecord.DebtRecord.UserCode.UserCode_code.Name",
3     "value" : "UserCode",
4     "comment" : "In the business entity 'ArrearsRecord', the 'UserCode'
              attribute in the 'ArrearsRecord' entity is associated with and
              brings out the name of the 'UserCode'."
5 }
```

As IDs are string values, if programmers call wrong or even non-existent IDs, compilers will not report the problems. When programs call the wrong IDs, they will fail at runtime. In this example, when querying the mobile number of a user, the `DebtRecordList_frm_extendConstruct.ts` file calls a wrong ID:

```
1 Screen() {  ...
2     var filter = '{" FilterField ": "UserCode.UserCode_CODE", "Value":"' +
            code + '", "Lbracket": null ," Rbracket": null , "Relation": 0, "
            Expresstype": 0," Compare": 0 }';
3     ...
4     return this . listDataService . filter ( filter , sort );
5 }
```

As IDs are case-sensitive, the above snippet retrieves an empty value and causes the bug.

C2.2 Corner Case (23/248, 9.27%). These bugs are caused by handling special cases, such as zero or null values, that the software does not properly anticipate. For example, in project ***mcf, a table should display a zero if a household has no previous meters, but this table fails to display anything. The buggy code is as follows:

```
1 List<IEntityData> entityDataList = lcp.query( entityFilter );
2 engineRootData.setValue("StartNum", (BigDecimal) entityDataList.get(0).
        getValue("startNum"));
```

Line 2 sets the value of this table. When a household has no previous meters, `entityDataList` is `null`, and calling the `get(0)` method throws an exception. As a result, the table fails to display this value. In this bug, programmers fail to handle the corner case, `null`.

C2.3 Other (10/248, 4.03%). This category includes other types of domain knowledge bugs like exception handling. For instance, in ***odm, `WFCompnentImp.java` has the following code:

```
1  public void writeDocOpinion (...) {
2   IBefSessionManager befSessionManager = SpringBeanUtils.getBean (...) ;...
3   befSessionManager.createSession () ;...
4   try {...
5       if ("Proofreader".equals(type)) {
6           writeCheckPerson(wfContext, type);
7           return;
8       }...
9     } catch (Exception e) {...
10    } finally {
11      if (! StringUtils .isEmpty(befSessionManager.getCurrentSessionId ())) {
12        befSessionManager.closeCurrentSession ();
13      }
14    }
15  }
```

In the above code, Line 3 creates a session, and Line 12 closes this session. However, if a person is a proofreader, Line 7 bypasses Line 14, and the session is not closed.

The above observations lead to a finding:

> **Finding 4.** 29.44% of LCA bugs are introduced due to the lack of domain-specific knowledge.

As domain-specific knowledge is valid across projects, it is feasible to implement tools to detect bugs for detecting ID mismatches and corner cases for all LCA.

In summary, 78.23% of LCA bugs are related to project-specific knowledge. Learning-based tools are suitable for detecting these bugs since they can learn project-specific knowledge. 29.43% of LCA bugs are related to domain-specific knowledge. Rule-based tools can be useful for detecting these bugs since their rules hold across projects.

C. RQ3. Repair

1) Protocol: This research question explores the modifications to repair LCA bugs. For each bug, we analyze the modifications of its bug-fixing commit. The modifications can be applied on source code, configurations, and other software artifacts. By analyzing commit histories and modification details, we categorize repairs and provide illustrative examples to demonstrate the details.

2) Result: We identify two categories of repairs.

R1 Modification to Code (219/248, 88.31%). This type of repair mainly modifies source files to fix bugs.

R1.1 Adjusting Functionality (115/248, 46.37%). Programmers modify code functionalities to fix bugs. For instance, in the ***odm project, a bug report complains that a table is unordered. Programmers introduce this bug because they forget to define the sorting criteria of this table. To fix this, programmers modify the functionality:

```
1  { ...
2    "Content" : {
3      "code" : "DepartmentGroupMobile",
4  +   "sorts" : [ {
5  +       "sortType" : 0,
6  +       "sortField" : "PRIORITY"
7  +     } ],  ...}
8  }
```

Lines 4 to 7 of the above code add a sorting method and rank the table by priority. To fix this bug, these code lines implement new functionality for this table.

R1.2 Aligning ID (40/248, 16.13%). If IDs are mismatched, programmers can align mismatched IDs. In Figure 4, a table fails to list mobile phone numbers due to mismatched IDs. To fix this bug, programmers made the following modification:

```
1  - filter += '{"FilterField": "UserCode.UserCode_CODE", "Value":...}';
2  + filter += '{"FilterField": "UserCode.UserCode_code", "Value":...}';
```

The programmer corrects the ID of the key used for the field filter to locate the corresponding data. As a result, we classify this as an aligning ID method.

R1.3 Changing Data Process (33/248, 13.30%). In this category, the bugs are caused by incorrect data processing methods, and the fix involves modifying the filter or calculation logic. For example, in the project ***mcf, a programmer encountered an internal bug where the total fee amount was incorrect. This bug arises because the data processing logic wrongly included other expenses in the total fee calculation. To fix the bug, the programmer made the following modification:

```
1    BigDecimal otherExpenses = this.sumAmount(otherExpenses);
2    BigDecimal feeSums = this.sumAmount(feeSums);
3  - feeSums = feeSums.add(otherExpenses);
```

The original code mistakenly excluded `otherExpenses` from the calculation of `feeSums`. In the snippet, two variables, `otherExpenses` and `feeSums`, are used to store the sum of other expenses and the sum of fees, respectively. The problematic line, denoted by the subtraction symbol (-), adds the `otherExpenses` to `feeSums`, which was incorrect according to the intended logic. The programmer removed this addition to correctly separate the two amounts, ensuring the total fee calculation does not erroneously include other expenses.

R1.4 Adjusting Automata (11/248, 4.44%). An automaton defines states, events, and transitions for complex behaviors. According to the current state of automata, programs can produce outputs for specific inputs. When bugs occur in this process, programmers can modify automata. For instance, to fix the bug in Figure 3, programmers modified the logic of state transitions, and we categorize this modification to adjusting the automata.

R1.5 Others (20/248, 8.06%) This category includes other types of modification to code. For instance, in project ***min, `DownloadPDFVOAction.java`, the programmer exchanges the method to throw an exception when failing to download the PDF of the receipt.

```
1  catch (Exception e) {
2  -   throw new RuntimeException(e);
3  +   throw new CommonException("***min", "DownloadError", "Download
       Error!" + e.getMessage()+e.getStackTrace(), null, ExceptionLevel.Info);
4  }
```

The above observations lead to a finding:

> **Finding 5.** Modifications on source files are essential in repairing 88.31% of LCA bugs.

R2 Modification to Customization (29/248, 11.69%). This type of repair modifies customizations to fix bugs. This repair method is closely related to the modification of UI settings. UI setting refers to the configuration and customization of the user interface, including adjusting the layout, design, components, and behavior. In LCA development, UI settings are a popular tool for ease of customization and application design without the need for extensive programming skills. This accessibility allows for faster development, as users can visually configure elements and see changes in real time.

R2.1 Changing Enable Conditions (20/248, 8.06%). Programmers can modify the conditions to enable or disable GUI elements when repairing bugs. For instance, as shown in Figure 7, after clicking the "Add" button to create a policy, this button should be disabled until the current policy is approved, but this "Add" button is never disabled. To fix this

bug, programmers modified the settings of this button, and the modification caused the following code modification:

```
1  "canAdd": {
2      "condition" : [
3          {"lBracket": "", "source": "state", "compare": "===", "target": "'
               init '", " relation ": "", "rBracket": ""}
4  -       ,{"lBracket": "", "source": "getData('{DATA^/root-component/
           billstate / billState }')", "compare": "===", "target": "' Billing '", "
           relation ": "or", "rBracket": ""}
5      ],...    }
```

In the above setting, the `canAdd` condition determines whether the "Add" button is enabled. In the buggy code, Line 3 checks whether a new policy is created, *i.e.*, whether `state` is `init`. Line 4 checks whether the state of the created policy is `Billing`. As the button is enabled when either condition is satisfied, the button is enabled when the policy is created. In the fixed code, programmers delete Line 4 and remove the unnecessary condition.

R2.2 Changing GUI Setting (9/248, 3.63%). As LCPS provides many GUI settings, programmers can modify the settings to repair bugs. For instance, in project ***odm, a bug report complains that the fields in a list do not wrap if the fields have long names. To fix this bug, programmers modify the settings of the list and cause the following code modification:

```
1  " fields ":[{
2      "caption" : "current  participant " ,...
3  +   "allowGrouping" : true
4  }
```

The above patch adds the `allowGrouping` to enable the wrapping of fields. The above observations lead to a finding:

> **Finding 6.** Changing enable conditions and GUI settings fix 8.06% and 3.62% of LCA bugs, respectively.

In summary, even in LCAs, repairing bugs mainly requires modifications to the source code. Only 11.69% of LCA bugs are fixed by customization modifications.

D. RQ4. Association

1) Protocol: In this RQ, we aim to explore the associations between symptoms, causes, and repair methods. Based on the results of previous RQs, we build a graph to denote the associations. To measure the strength of these associations, we apply the *lift* metric [31], which quantifies the relationship between two categories as follows:

$$lift(A, B) = \frac{P(A \cap B)}{P(A) \times P(B)} \quad (1)$$

In this equation, $P(A)$, $P(B)$, and $P(A \cap B)$ correspond to the probabilities that a bug falls under category A, category B, and both categories A and B, respectively. A *lift* value measures how strongly two categories are associated, but a higher *lift* value does not necessarily imply a larger proportion of bugs. Typically, a *lift* value greater than one indicates that there is a meaningful association between A and B. However, to increase the robustness of the associations we identify, we define a stronger threshold: An association is only considered valid if the *lift* value exceeds 1.5 and the intersection between A and B involves at least five bug cases. In this context, we focus specifically on analyzing the symptom-to-cause and cause-to-repair associations.

2) Result: In Figure 8, each column denotes a category of symptoms, causes, and repairs. In each column, a thin bar denotes the percentage of this category. The edges between the two columns indicate their association. The thickness of an edge is proportional to the strength of the association, measured by the *lift* value. As defined in Equation 1, thicker edges denote higher *lift* values and stronger associations. For instance, as shown in Figure 8, $P(C1.3)$ is 13.70%; $P(R2.1)$ is 8.06%; and $P(C1.3 \cap R2.1)$ is 2.82%. We calculate the lift value between C1.3 and R2.1 as $\frac{2.82\%}{13.70\% \times 8.06\%} = 2.55$. As the lift value is more than 1.5, we consider that there is a strong association between C1.2 (GUI designs) and R2.1 (changing GUI settings).

Figure 8 shows that the associations between symptoms and causes are weak. The links between symptoms and causes are fewer than those between causes and repairs, and the links are lighter. The result indicates that the same symptoms can be triggered by the lack of both project-specific knowledge and domain-specific knowledge. As an extreme case, wrong process bugs are caused by mismatches (10), data retrieval (7), corner cases (5), and logical errors (31). This demonstrates that bugs in interaction flow could stem from multiple underlying causes, requiring a holistic approach to diagnose and fix them effectively. These weak associations between symptoms and causes bring challenges to the diagnosis of LCA bugs. As a comparison, the association between causes and repairs tends to be stronger since there are more links, and links are thicker. For instance, adjusting automata repairs most incorrect automata. The above observations lead to a finding:

> **Finding 7.** In LCAs, the same symptoms are caused by multiple reasons, but when the reasons are found, most repairs are strongly associated with causes.

E. Threats to Validity

The internal validity includes manual errors in our study. This threat could further affect the links between symptoms and causes. To reduce the threat, we carefully discussed and refined the categories, and asked industrial professional programmers to check our results. The second threat is related to the completeness of our taxonomy. Like other studies, our built taxonomy is based on observations. This threat could be mitigated if researchers collect more data from other sources.

The external validity includes our selected subjects. We are the first to analyze LCA bugs. As it is difficult to collect LCAs for analysis, our subjects lack diversity. They are developed by an LCDP and collected from the same company. The results could be different if follow-up researchers analyze more LCAs from other sources, *e.g.*, mobile LCAs. This threat could be further reduced if researchers can collect and analyze open-source LCAs in the future. Another external threat is

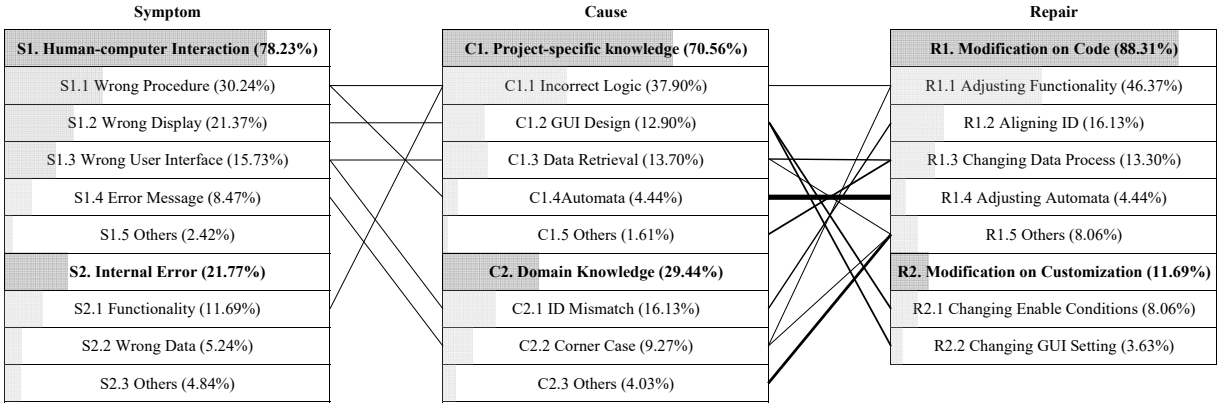


Fig. 8. The distributions and associations of symptoms, causes, and repairs.

that all analyzed bugs originate from a single company. This threat could be reduced if other researchers collect low-code application bugs from more companies.

Although our inspected bugs are comparable to prior studies [23], [24], we have inspected a limited number of bugs. This threat is shared by all empirical studies if their analysis is manual. Due to the huge effort of manual analysis, researchers can analyze only limited subjects manually. As another shared external validity, all empirical studies are a bite of time. With the evolution of LCAs and LCDPs, future data can illustrate other characteristics. As low-code development is a recent trend, although we tried to find the LCAs with the longest maintenance histories, most of the subjects have started since 2023. It can be interesting to replicate our study when LCAs accumulate more bugs.

V. Interpretation of Our Findings

In this section, we interpret our findings:

Integrating LCDPs with advanced techniques like GUI testing. Although we do not analyze LCDP bugs, our study provides feedback from LCDP users, and our findings are useful for improving LCDPs. For instance, as most LCA symptoms are related to human-computer interactions (Finding 1), LCDPs can integrate more GUI testing tools. Furthermore, as we report the associations among symptoms, causes, and repairs (Finding 7), LCDP can offer better support for diagnosing bugs.

Identifying test oracles for LCA bugs. Finding 2 shows that most LCA bugs are not crashes. Finding 3 shows that most bugs are specific to projects. To detect these bugs, researchers can learn test oracles if they mine project-specific rules. These tools are specific to projects since they require project-specific knowledge. Meanwhile, Finding 4 shows that 29.44% of LCA bugs are introduced due to the lack of domain-specific knowledge. If domain-specific knowledge is encoded, tools can detect bugs that are general and appear in multiple LCAs. Researchers need to identify more complicated test oracles when detecting LCA bugs.

Programming expertise for implementing LCAs. Many practitioners and researchers believe that low-code development allows non-technical people to create LCAs. Our findings are mixed. On one hand, as shown in Table I, with the support of LCPDs, only several programmers can write large LCAs in a short period of time. On the other hand, repairing most LCA bugs still requires code modifications (Finding 5). It still needs much programming expertise to identify and repair LCA bugs. Therefore, while low-code development lowers the barrier to software creation and allows broader participation from non-technical people, professional programmers remain essential, especially when diagnosing and repairing bugs.

VI. Related Works

Empirical studies on low-code development platforms. Researchers have conducted some empirical studies to understand these platforms. Sahay *et al.* [8] compare the features and functionalities of eight LCDPs. Bock and Frank *et al.* [9] conducted an exploratory study on seven different LCDPs, aiming to identify the essence of low-code development. Besides the characteristics of LCDPs, some studies explore the programmers' perspective. Alsaadi *et al.* [32] conducted a survey study to understand the factors for the utilization of LCDPs. Luo *et al.* [10] conducted a study on the challenges of low-code development. Al Alamin *et al.* [11] analyze how programmers migrate to LCDPs. Al Alamin *et al.* [12] analyze how programmers discuss the adoption and barriers of LCDPs. Other studies like Liu *et al.* [33] analyze the bugs of LCDPs. Although they support the development of LCAs, LCDPs themselves are not LCAs. We are the first to analyze the characteristics of bugs in LCA.

Empirical studies on bugs in specific software systems. Researchers have conducted empirical studies to analyze bugs in specific software. Thung *et al.* [17] analyze bugs in machine learning systems. Sun *et al.* [34] analyze bugs in machine learning programs. Besides machine learning programs, Zhang *et al.* [19] and Jia *et al.* [20] analyze bugs in Tensorflow. Sun *et al.* [34] analyze bugs in more deep-learning libraries. Tambon *et al.* [35] analyze silent bugs in Keras and TensorFlow. Zhang *et al.* [36] analyze job failures in deep learning. Besides machine learning bugs, researchers [37], [23], [38], [39], [40] have conducted various empirical studies to understand compiler bugs. These studies have derived interesting findings about the buggy locations [38], repair patterns [38], [37], [40],

and causes [38], [39] of compiler bugs. Besides the above-mentioned bugs, researchers analyze bugs in other types of software *e.g.*, printed circuit board bugs [41] and webassembly runtime bugs [42]. We are the first to analyze bugs in LCA, complementing the prior studies.

End user programming. End user programming refers to a set of methods, techniques, and tools that enable nonprofessional software developers to create, modify, and extend a software artifact [43], [44]. End user programming is a hot research topic, and the related work can be traced back to the 1990s [45], [46]. Lieberman *et al.* [47] classify different types of techniques for supporting end user programming. Kelleher *et al.* [48] present a taxonomy of languages and environments of end user programming. Researchers have analyzed various perspectives of end user programming, including daily activities [49], science data processing [50], and MATLAB script programming [51]. As a recent trend, end user programming has been introduced to the AI domain, like AI model development [52], interaction design [53], and education and teaching [54]. As low-code programming is a type of end user programming, the findings in our study are useful for understanding bugs in end user programming.

VII. Conclusion and Future Work

Low-code development has attracted much attention from academics and industries since this new programming paradigm lowers the bar of programming. To understand this emerging paradigm, researchers have conducted various studies on LCPD bugs and how programmers perceive low-code development. However, as LCAs have business values, few LCAs are open source. As a result, to the best of our knowledge, no study has explored LCA bugs. From our industrial partner, we collected more than two hundred LCA bugs and conducted the first empirical study on commercial LCA bugs. Our study covers the symptoms, causes, and repairs of LCAs and derives seven findings.

In future work, our study can be extended in the following perspectives. First, it is worth exploring which components of LCPDs are particularly vulnerable to introducing LCA bugs. Second, the correlations between LCA bugs and LCPD bugs are worth further investigation. Finally, it is worth exploring how to effectively detect and repair LCA bugs.

Acknowledgment

We appreciate reviewers for their insightful comments. This work is sponsored by National Key R&D Program of China No. 2023YFB4503804 and National Natural Science Foundation of China No. 62272295.

References

[1] R. Waszkowski, "Low-code platform for automating business processes in manufacturing," *IFAC-PapersOnLine*, vol. 52, no. 10, pp. 376–381, 2019.

[2] K. S. Koong, L. C. Liu, and X. Liu, "A study of the demand for information technology professionals in selected internet job portals," *Journal of Information Systems Education*, vol. 13, no. 1, pp. 21–28, 2002.

[3] J. F. Pane and B. A. Myers, "More natural programming languages and environments," in *End user development*, 2006, pp. 31–50.

[4] C. Richardson, J. R. Rymer, C. Mines, A. Cullen, and D. Whittaker, "New development platforms emerge for customer-facing applications," *Forrester: Cambridge, MA, USA*, vol. 15, 2014.

[5] J. R. Rymer, R. Koplowitz, S. Leaders, K. Mendix, S. Leaders, G. ServiceNow, S. Performers, W. MatsSoft, and T. Contenders, "The forrester wave™: low-code development platforms for ad&d professionals," *Em The*, vol. 13, 2019.

[6] J. Wong, M. Driver, and P. Vincent, "Low-code development technologies evaluation guide," *Gartner. Verfügbar unter: https://www.gartner. com/en/documents/3902331/low-code-development-technologies-evaluation-guide (Zugriff am: 15.10. 2020)*, 2019.

[7] Mendix, "A survey of the low-code market," 2025. [Online]. Available: https://www.mendix.com/blog/low-code-market/

[8] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *Proc. SEAA*, 2020, pp. 171–178.

[9] A. C. Bock and U. Frank, "In search of the essence of low-code: an exploratory study of seven development platforms," in *Proc. MODELS-C*, 2021, pp. 57–66.

[10] Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, "Characteristics and challenges of low-code development: the practitioners' perspective," in *Proc. ESEM*, 2021, pp. 1–11.

[11] M. A. Al Alamin, S. Malakar, G. Uddin, S. Afroz, T. B. Haider, and A. Iqbal, "An empirical study of developer discussions on low-code software development challenges," in *Proc. MSR*, 2021, pp. 46–57.

[12] M. A. A. Alamin, G. Uddin, S. Malakar, S. Afroz, T. Haider, and A. Iqbal, "Developer discussion topics on the adoption and barriers of low code software development platforms," *Empirical software engineering*, vol. 28, no. 1, p. 4, 2023.

[13] "Exploring low-code development: A comprehensive literature review," Oct. 2023. [Online]. Available: https://csimq-journals.rtu.lv/csimq/article/view/csimq.2023-36.04

[14] "Appsmith: Deliver custom ai-powered apps and agents faster," https://www.appsmith.com/, 2024.

[15] "Lesson 3 - now code it," https://docs.appsmith.com/getting-started/tutorials/the-basics/write-js-code, 2024.

[16] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021.

[17] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *Proc. 23rd ISSRE*, 2012, pp. 271–280.

[18] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: a large-scale empirical study," in *Proc. MSR*, 2017, pp. 413–424.

[19] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proc. 27th SIGSOFT*, 2018, pp. 129–140.

[20] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside tensorflow," in *Proc. 25th DASFAA*, 2020, pp. 604–620.

[21] A. Rahman and E. Farhana, "An exploratory characterization of bugs in covid-19 software projects," 05 2020.

[22] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: An empirical study of bug characteristics in modern open source software," 10 2006, pp. 25–33.

[23] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *Proc. ASE*, 2021, pp. 42–54.

[24] X. Wu, J. Yang, L. Ma, Y. Xue, and J. Zhao, "On the usage and development of deep learning compilers: an empirical study on tvm," *Empirical Software Engineering*, vol. 27, no. 7, p. 172, 2022.

[25] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of the international multiconference of engineers and computer scientists*, vol. 1, no. 6, 2013, pp. 380–384.

[26] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[27] J. Kohen, "A coefficient of agreement for nominal scale," *Educ Psychol Meas*, vol. 20, pp. 37–46, 1960.

[28] Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, and X. Shi, "Analyzing bug fix for automatic bug cause classification," *Journal of Systems and Software*, vol. 163, p. 110538, 2020.

[29] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying the root cause of bugs," *arXiv preprint arXiv:1907.11031*, 2019.

[30] D. Wang, S. Li, G. Xiao, Y. Liu, and Y. Sui, "An exploratory study of autopilot software bugs in unmanned aerial vehicles," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3468264.3468559

[31] J. Han, M. Kamber, and J. Pei, "Data mining: Concepts and techniques," 2011.

[32] H. A. Alsaadi, D. T. RADAIN, M. M. Alzahrani, W. F. Alshammari, D. Alahmdi, and B. Fakieh, "Factors that affect the utilization of low-code development platforms: survey study." *Romanian Journal of Information Technology & Automatic Control/Revista Română de Informatică și Automatică*, vol. 31, no. 3, 2021.

[33] D. Liu, H. Jiang, S. Guo, Y. Chen, and L. Qiao, "What's wrong with low-code development platforms? an empirical study of low-code development platform bugs," *IEEE Transactions on Reliability*, 2023.

[34] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *Proc. 24th APSEC*, 2017, pp. 348–357.

[35] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, "Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow," *Empirical Software Engineering*, vol. 29, no. 1, p. 10, 2024.

[36] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proc. ICSE*, 2020, pp. 1159–1170.

[37] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *Journal of Systems and Software*, vol. 174, p. 110884, 2021.

[38] Z. Wang, D. Bu, A. Sun, S. Gou, Y. Wang, and L. Chen, "An empirical study on bugs in python interpreters," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 716–734, 2022.

[39] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proc. ESEC/FSE*, 2021, pp. 968–980.

[40] C. Zhang, B. Chen, L. Chen, X. Peng, and W. Zhao, "A large-scale empirical study of compiler errors in continuous integration," in *Proc. ESEC/FSE*, 2019, p. 176–187.

[41] X. Zhao, H. Jiang, S. Guo, D. Liu, H. Liu, C. Shi, and X. Li, "A comprehensive study of open-source printed circuit board (pcb) design software bugs," *IEEE Transactions on Instrumentation and Measurement*, 2024.

[42] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A comprehensive study of webassembly runtime bugs," in *Proc. SANER 2023*, 2023, pp. 355–366.

[43] M. M. Burnett. and C. Scaffidi, *The Encyclopedia of Human-Computer Interaction, 2nd Ed.* The Interaction Design Foundation, 2013.

[44] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, 2011.

[45] B. A. Nardi, *A small matter of programming: perspectives on end user computing*. MIT press, 1993.

[46] M.-F. Costabile, D. Fogli, C. Letondal, P. Mussio, and A. Piccinno, "Domain-Expert Users and their Needs of Software Development," in *Proc. HCI 2003*, 2003. [Online]. Available: https://hal.science/hal-01299738

[47] H. Lieberman, F. Paternò, M. Klann, and V. Wulf, *End-User Development: An Emerging Paradigm*. Springer Netherlands, 2006.

[48] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *CSUR*, vol. 37, no. 2, pp. 83–137, 2005.

[49] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the numbers of end users and end user programmers," in *Proc. VL/HCC 2005)*. IEEE, 2005, pp. 207–214.

[50] G. Fischer, K. Nakakoji, and Y. Ye, "Metadesign: Guidelines for supporting domain experts in software development," *IEEE software*, vol. 26, no. 5, pp. 37–44, 2009.

[51] N. Gulley, "Improving the quality of contributed software and the matlab file exchange," in *Proc. 2nd WEUSE*, 2006, pp. 8–9.

[52] K. Kahn, R. Megasari, E. Piantari, and E. Junaeti, "Ai programming by children using snap! block programming in a developing country," in *Proc. EC-TEL*, vol. 11082, 2018.

[53] A. Bunt, C. Conati, and J. McGrenere, "Mixed-initiative interface personalization as a case study in usable ai," *AI Magazine*, vol. 30, no. 4, pp. 58–58, 2009.

[54] F. Paternò, "teaching end-user development in the time of iot and ai," in *Proc. IFIP*. Springer, 2021, pp. 257–269.