nemo

UFES

JÚLIA S. BORGES
MONALESSA P. BARCELLOS

Causal Loop Diagram in Software Engineering: Theoretical Foundations and Practical Guidelines

2026

PREFACE

Software engineering projects often involve many interconnected elements, such as people, processes, tools, and technical decisions. Because of these interactions, problems in software engineering are not always caused by a single factor. Instead, they often emerge from relationships between different parts of the system. System Thinking (ST) offers a way to better understand this complexity by focusing on how elements of a system influence one another over time.

In the ST context, a system (e.g., a software organization) consists of elements and interconnections (e.g., in a software organization there is a relation between the development team, the software artifacts it produces, and the policies that influence their production) coherently organized in a structure that produces a characteristic set of behaviors (e.g., the development team produces software to accomplish the team function in the organization).

One modeling tool commonly used in ST is the Causal Loop Diagram (CLD). A CLD is a visual representation that shows cause-and-effect relationships between variables (i.e., elements) in a system. By connecting these variables, CLDs help to understand how the system behavior evolves.

In Software Engineering, CLDs can help analyze complex situations such as delays in software projects, increasing technical debt, communication challenges within teams, or recurring quality issues. By visually representing variables and relationships, CLDs help software engineers explore possible causes of problems and better understand the dynamics of software development environments.

The objective of this document is to offer an introduction to CLD for software engineers. The goal is to provide a theoretical understanding of the main CLD elements and support in applying CLD to address software engineering problems.

The document is organized into three parts. First, we introduce CLD concepts, along with examples. Second, we present a step-by-step guide on how to create a CLD. Finally, we introduce CaLMo, a web tool designed to support CLD creation and analysis.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Monalessa Barcellos, for her continuous guidance, encouragement, and valuable insights throughout this work. I am also thankful to the undergraduate students Thiago Felippe Neitzke Lahass and Amanda Brito Apolinário, who actively contributed to the development of CaLMo. Thiago also collaborated on the theoretical foundation about archetypes in this document. Their collaboration and commitment were essential to this work.

Júlia S. Borges

This work was supported by the Espírito Santo Research and Innovation Support Foundation - FAPES (Processes 2023-5L1FC, 2022-NGKM5, 2021-GL60J, and T.O.1022/2022), the National Council for Scientific and Technological Development - CNPq (Process 304787/2025-6), and the Coordination for the Improvement of Higher Education Personnel - CAPES (Finance Code 001).

CONTENTS

PART I: FOUNDATIONS..... 5
1. GETTING STARTED WITH SYSTEM THINKING..... 6
1.1. WHAT IS A SYSTEM?..... 6
1.2. WHAT IS SYSTEM THINKING?..... 7
1.3. WHAT IS A CAUSAL LOOP DIAGRAM?..... 7
1.4. ELEMENTS OF A CAUSAL LOOP DIAGRAM..... 8
1.4.1 VARIABLE..... 9
1.4.2 CAUSAL LINK..... 9
1.4.3 DELAY..... 11
1.4.4 FEEDBACK LOOP..... 12
1.5 ARCHETYPES..... 14
Fixes That Fail..... 15
Shifting the Burden..... 16
Limits to Success..... 17
Drifting Goals..... 18
Growth and Underinvestment..... 20
Success to the Successful..... 22
Escalation..... 23
Tragedy of the Commons..... 25
PART II: PRACTICAL GUIDANCE..... 27
2. BUILDING A CLD STEP BY STEP..... 28
2.1 ILLUSTRATIVE APPLICATION SCENARIO..... 28
2.2 STEP BY STEP..... 28
Step 1: Identify the System and the Context..... 29
Step 2: Identify Variables..... 31
Step 3: Identify Causal Links and Delay..... 32
Step 4: Explore Loops and Archetypes..... 34
PART III: SUPPORTING TOOL..... 37
CaLMo: Causal Loop Diagram Modeler (Apolinário et al., 2025)..... 38
3.1 Introducing CaLMo..... 38
3.2 CaLMo Features..... 39
3.2 CaLMo Limitations and Future Improvements..... 45
FINAL CONSIDERATIONS..... 46

PART I: FOUNDATIONS

1. GETTING STARTED WITH SYSTEM THINKING

Software organizations involve several processes, people, practices, culture, and other factors that affect their behavior. Understanding the organizational environment is crucial for improving processes and products. **System Thinking** can help in this matter. It views an organization as a **system** composed of elements and interconnections coherently organized in a structure that produces a characteristic set of behaviors. System Thinking offers several tools, among which the **Causal Loop Diagram** stands out. It represents feedback-driven behaviors in complex systems and enables the visualization of organizational dynamics and the identification of systemic patterns. By understanding how the organization behaves, it is possible to identify problems and define actions aimed at improving products and processes (APOLINÁRIO et al., 2025).

In this section, we introduce key concepts related to System Thinking and Causal Loop Diagram. Each concept is explained with examples to help understand it in the Software Engineering context.

1.1. WHAT IS A SYSTEM?

A **system** consists of a set of interconnected elements organized in a structure that produces characteristic behaviors (MEADOWS, 2008); STERMAN, 2010). For example, an organization (or a part of it) is a system. Its elements may include people, processes, tools, and decisions (among others) that work together (i.e., relate to each other) and influence how the system behaves. Figure 1 shows us an example of a system in Software Engineering.

In the Software Engineering context, a development team is an example of a system. In the team, developers, project managers, development practices, tools, deadlines, and quality requirements (among others) are all connected. A change in one element can influence others. For instance, tight Deadlines may increase Development Pressure, which can lead to more Bugs and affect (decrease) Code Quality, in turn, increases Rework, which further elevates Development Pressure. Viewing the development environment as a system helps better understand these relationships and the dynamics that shape project outcomes.

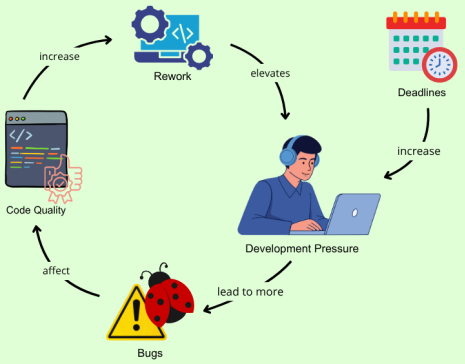

Figure 1. Systems overview in the Software Engineering context.

1.2. WHAT IS SYSTEM THINKING?

System Thinking is an approach used to understand complex situations by focusing on the relationships and interactions between elements of a system, rather than analyzing each element separately (FATEMA & SAKIB, 2017). Because it focuses on how elements influence one another, System Thinking helps people better understand how a system works. This perspective can support the identification of patterns, the anticipation of system behavior, and the search for possible improvements (KIM, 2018).

To support this understanding, System Thinking offers modeling tools that help represent systems visually. These tools create simplified models that make relationships and structures easier to understand (FAKHIMI, ROBERTSON, & BONESS, 2017). One of these tools is the **Causal Loop Diagram** (CLD), which is used to represent cause-and-effect relationships and feedback loops within a system. In this document, we will introduce CLDs and exemplify how they can be used to represent and help analyze situations in Software Engineering.

1.3. WHAT IS A CAUSAL LOOP DIAGRAM?

A **Causal Loop Diagram** (CLD) is a visual way to represent how different elements of a system influence one another. It is composed of variables, which

represent elements of the system, and connections, which show how a change in one variable can influence another (STERMAN, 2010).

In a CLD, arrows are used to connect variables and represent cause-and-effect relationships. This allows for visualizing how changes in one part of a system may affect other parts. Because of this, CLDs are widely used in System Thinking to explore complex situations and better understand how different elements interact. They help people analyze problems, identify possible causes, and reflect on how actions in one part of a system may influence other parts.

In Software Engineering, CLD can be used, for example, to represent relationships between factors that influence the development process, such as workload, code quality, and the number of bugs reported. Figure 2 illustrates an example of CLD. In the figure, ellipses represent variables, arrows represent relationships between variables, and signals + or - indicate how the variables influence each other. These elements will be explained in the following sections.

Consider, for example, a software development team working under tight deadlines. *Time pressure* leads the team to adopt *quick solutions* to deliver faster. Over time, this increases *technical debt*, which reduces *development speed* and lowers *code quality*. As a result, more *bugs* appear, leading to increased *rework*, which further reduces *development speed* and increases *time pressure.*

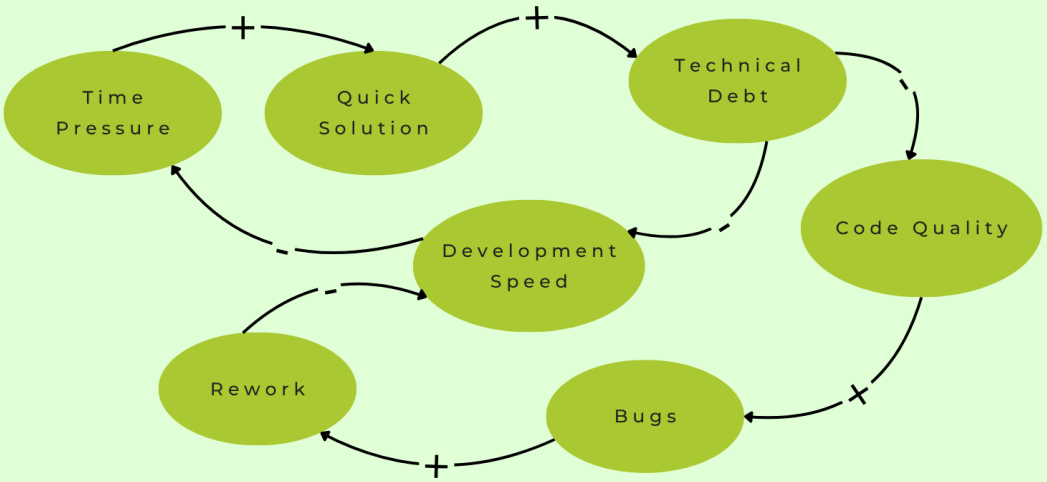

Figure 2. An example of a CLD in Software Engineering.

1.4. ELEMENTS OF A CAUSAL LOOP DIAGRAM

A CLD comprises elements that represent relationships within a system. The main elements are **variables**, **causal links**, and **feedback loops**. By combining these elements, a CLD allows for visualizing how different parts of a system influence one another and how changes in one variable can affect others over time. From these elements, it is possible to identify **archetypes**, which represent common behavior patterns that help identify actions to improve the system.

1.4.1 VARIABLE

A **variable** represents any factor that can change and influence the behavior of a system. In a CLD, each variable has a name that clearly identifies it. Giving each variable a unique name helps avoid confusion and makes the relationships in the diagram easier to understand. In a CLD, variables are represented in ellipses. In this document, we use *italic* to distinguish variables in the text. Figure 3 illustrates examples of variables in the software development context. These variables can influence others and change over time, affecting the system (e.g., a project) behavior.

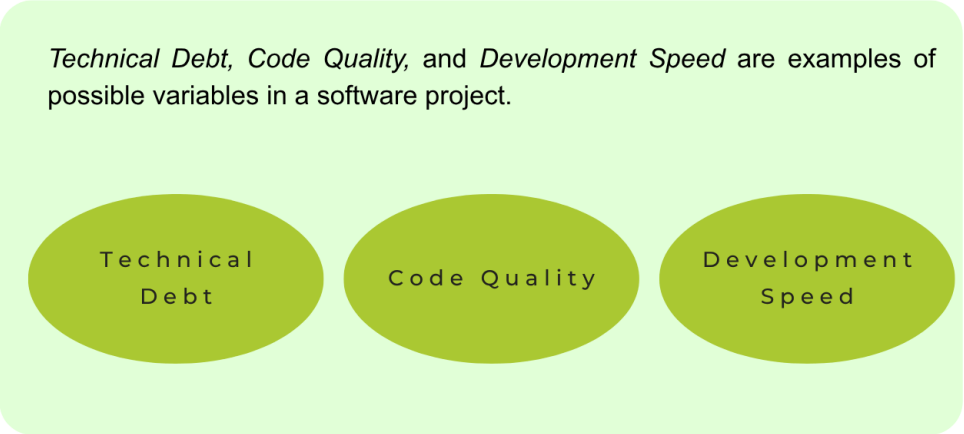

Figure 3. An example of Variables.

1.4.2 CAUSAL LINK

A **causal link** represents the connection between two variables in a system. It shows how a change in one variable can influence another. In a CLD, these connections are represented by arrows indicating the influence direction. The variable at the beginning of the arrow is the *source,* while the variable at the end

of the arrow is the *target.* Figure 4 illustrates an example of a causal link between two variables.

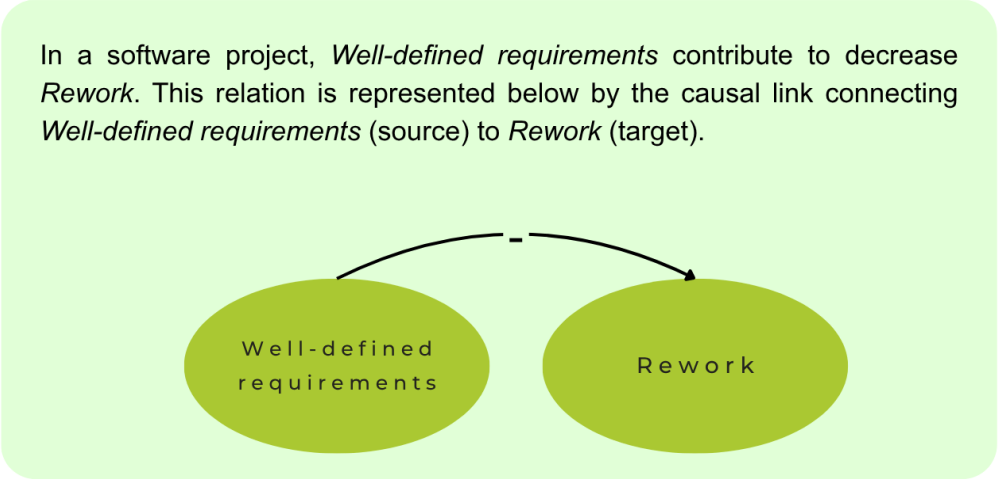

Figure 4. An example of Causal Link.

Note: A causal link always connects two variables (Figure 5). Thus, in a CLD, a variable cannot be the source and the target of the same causal link (i.e., a variable cannot be related to itself).

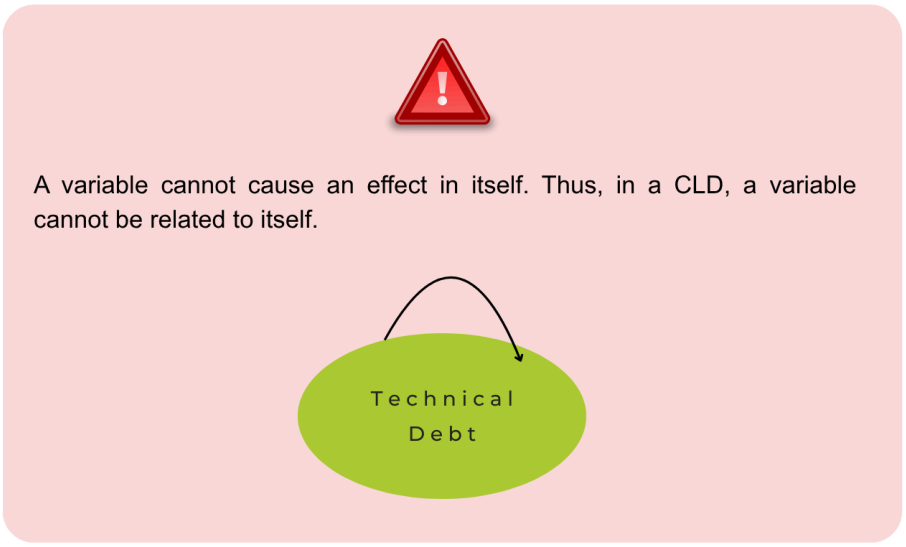

Figure 5. Variables cannot be self-related in a CLD.

Causal links can represent two types of influence (i.e., polarity): positive and negative.

A **positive causal link** shows that two variables change in the same direction. This means that when one variable (source) increases, the other (target) also increases. In the same way, when one variable (source) decreases, the other (target) also decreases (STERMAN, 2010). The causal link positive polarity is indicated by "+" in the arrow. Figure 6 illustrates an example of a positive causal link between two variables.

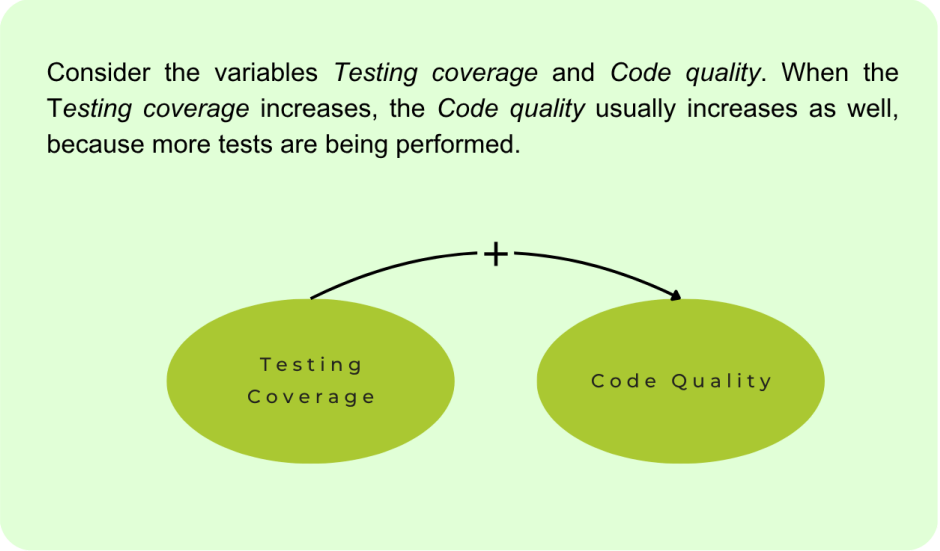

Figure 6. An example of Positive Causal Link.

A **negative causal link** shows that two variables change in opposite directions. This means that when one variable (source) increases, the other (target) decreases. In the same way, when one variable (source) decreases, the other (target) increases (STERMAN, 2010). The causal link negative polarity is indicated by "-" in the arrow. Figure 7 shows an example of a negative causal link between two variables.

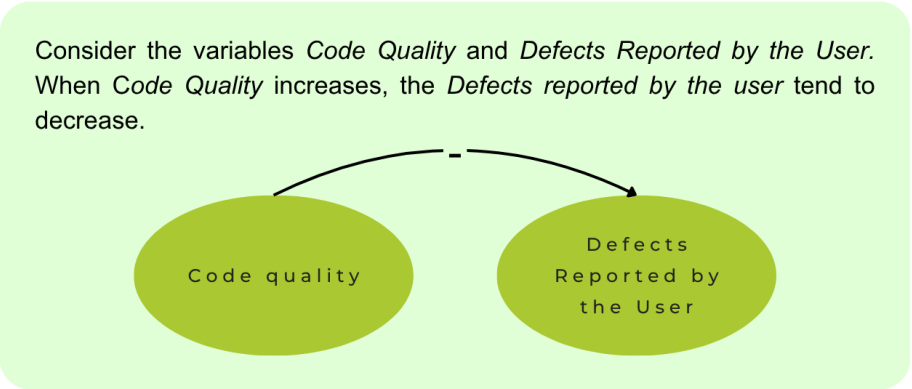

Figure 7. An example of Negative Causal Link.

Note: If a variable A (source) influences a variable B (target) in a positive way, it is not possible that variable A (source) influences variable B (target) negatively. In other words, if there is a positive causal link between A (source) and B (target), it is not possible to have a negative causal link between A (source) and B (target), and vice versa. Figure 8 illustrates this restriction.

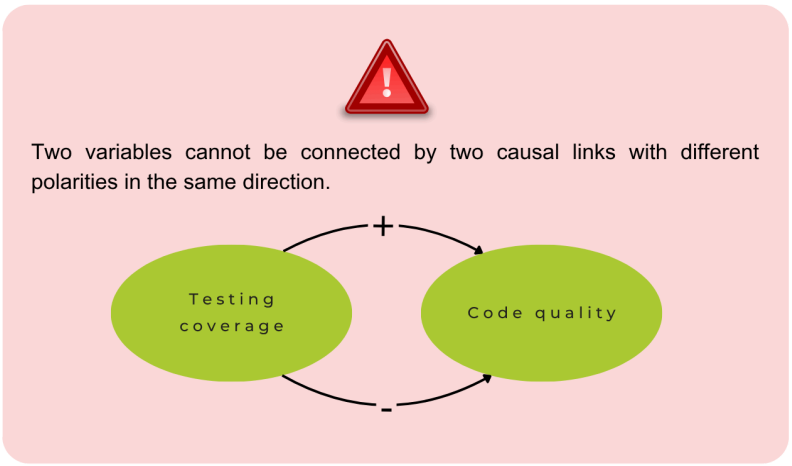

Figure 8. A causal link can have only one polarity.

1.4.3 DELAY

A **delay** represents a situation in which the effect of a change in one variable does not appear immediately in another variable. Instead, it takes some time – the delay -- before the impact becomes visible (MEADOWS, 2008). Delay is represented by two small lines that “cut” the causal link. Figure 9 illustrates an example of a relationship between two variables where the effect of the source variable takes time to be perceived in the target variable.

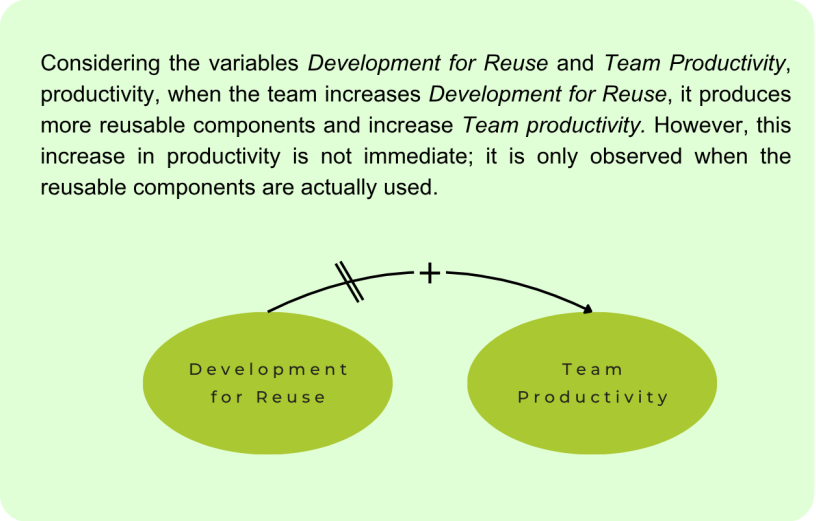

Figure 9. *An example of* Delay.

1.4.4 FEEDBACK LOOP

A **feedback loop** happens when variables are connected in a way that forms a closed cycle. This means one variable affects another, and eventually, the effect comes back to influence the original variable (MEADOWS, 2008; STERMAN, 2010). The simplest feedback loop has two variables connected in a closed cycle (as the example illustrated in Figure 10). However, feedback loops can involve multiple variables connected by multiple causal links, thereby forming a more elaborate closed cycle.

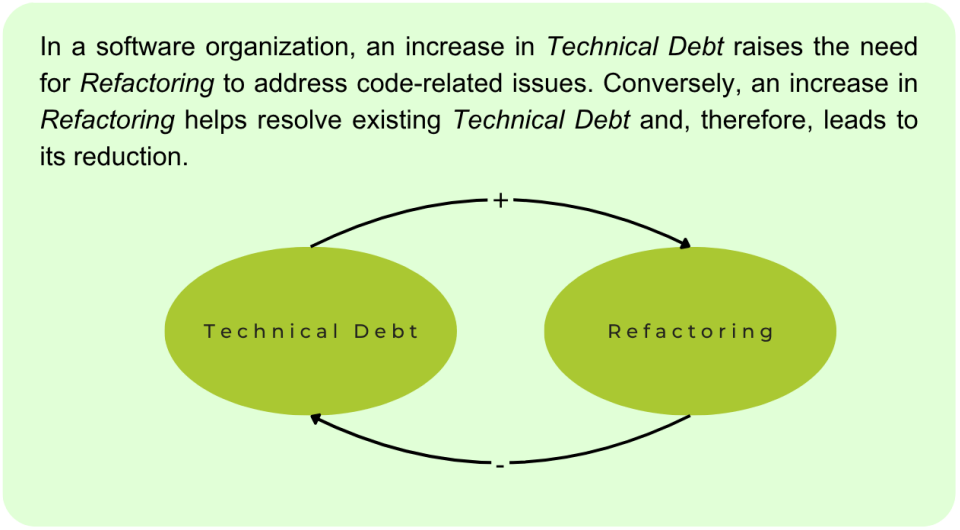

Figure 10. An example of Feedback Loop.

There are two types of feedback loop: balancing and reinforcing

A **reinforcing feedback loop** happens when a change in one variable increases the effect, and this effect further increases the original cause, creating a self-reinforcing cycle. As a result, the behavior tends to grow or decline rapidly over time because the loop keeps amplifying itself (MEADOWS, 2008).

Structurally, a reinforcing loop has an even number of negative causal links, including the possibility of none, which keeps the overall effect reinforcing (STERMAN, 2010). Reinforcing balance loops are indicated with “R” in the CLD. Figure 11 presents an example of a reinforcing loop composed of five variables, one positive causal link, and four negative causal links.

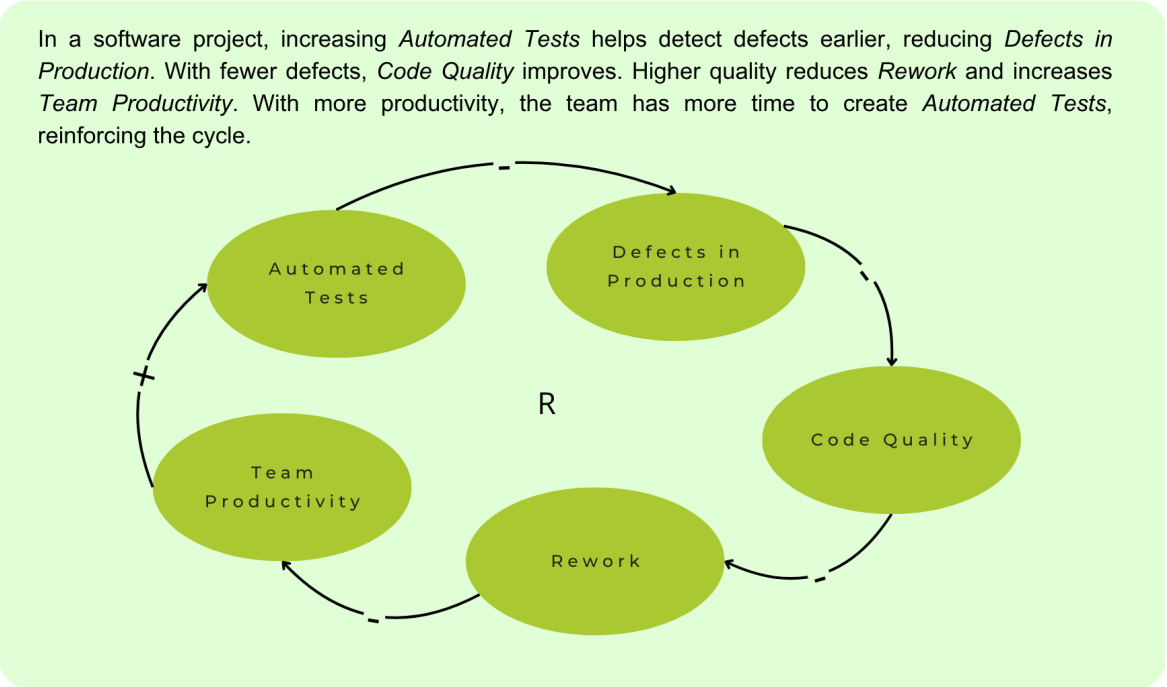

Figure 11. An example of Reinforcing Feedback Loop.

A **balancing feedback loop** aims to stabilize a system and keep it close to a desired state. It works by counteracting changes that move the system away from equilibrium (MEADOWS, 2008). In this type of loop, the effect reduces or limits the initial change, helping the system remain stable.

Structurally, a balancing loop has an odd number of negative causal links, which reverses the direction of the feedback (STERMAN, 2010). Balancing loops are indicated with “B” in the CLD. Figure 12 shows an example of the most basic form of a balancing feedback loop, which is composed of two variables, one positive causal link, and one negative causal link.

Consider the variables *Defects in Software Artifacts* and *Software Quality Techniques*. When *Defects in Software Artifacts* increase, the use of *Software Quality Techniques* also increases to address the problem (positive causal link). As the use of *Software Quality Techniques* increases, *Defects in Software Artifacts* decrease (negative causal link). This creates a balancing feedback loop that stabilizes the system over time.

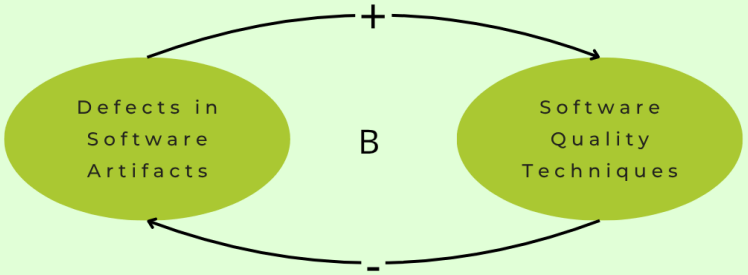

Figure 12. An example of Balancing Feedback Loop.

1.5 ARCHETYPES

One of the benefits of using CLDs is the possibility of identifying archetypes. An **archetype** is a structure that represents a common pattern of behavior in a system. It is composed of variables and causal links organized into feedback loops. Typically, an archetype includes at least two feedback loops working together.

Archetypes help identify recurring system behaviors and make it easier to understand, anticipate, and manage system dynamics. This document presents eight archetypes: **Fixes that Fail, Shifting the Burden, Limits to Success, Drifting Goals, Growth and Underinvestment, Tragedy of the Commons, Success to the Successful,** and **Escalation**. They are presented in a sequence that highlights their relationships. **Fixes that Fail** introduces immediate solutions to visible problems. **Shifting the Burden** shows the risk of relying on symptomatic solutions instead of addressing root causes. **Limits to Success** explains growth constrained by system limits. **Drifting Goals** illustrates how the gap is closed by lowering performance standards. **Growth and Underinvestment** extends Limits to Success by considering delays and

capacity erosion. The **Tragedy of the Commons** represents a situation where multiple actors overuse a shared limited resource, degrading the system over time. The **Success to the Successful** archetype describes a dynamic in which resources are distributed unevenly among two or more competing parties, favoring those that initially achieve greater success, and **Escalation** explains competitive and multi-actor dynamics.

Next, we introduce these archetypes, illustrate their modeling pattern, and present examples of their occurrence in the software engineering context. The definitions and modeling patterns are based mainly on (Kim; Anderson, 1998) and (MEADOWS, 2008).

Fixes That Fail

The **Fixes that Fail** archetype describes situations where a problem is addressed with a quick solution that provides immediate relief. However, this solution creates unintended consequences over time. These consequences bring the original problem back, sometimes even worse, leading to repeated use of the same fix and creating a cycle of recurring failures.

Structurally, this archetype includes two feedback loops: a balancing loop (B), which shows the short-term improvement, and a reinforcing loop (R), which represents the unintended consequences that worsen the problem after a delay. Figure 13 illustrates the modeling pattern of the Fixes that Fail archetype.

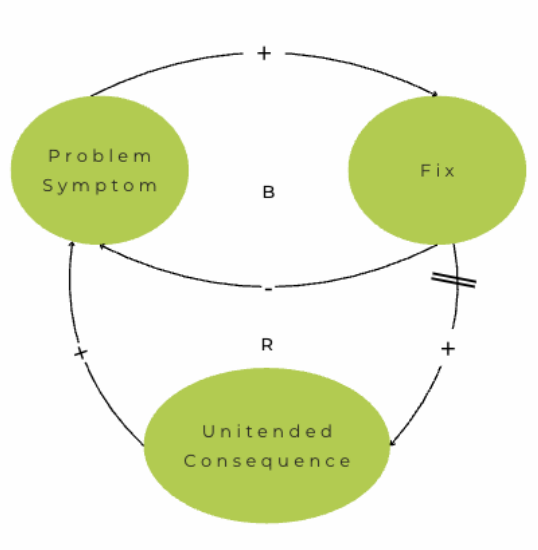

Figure 13. Fixes That Fail modeling pattern.

Figure 14 presents an example of the occurrence of the Fixes that Fail archetype.

Consider a software development team facing a *Late Schedule* (problem symptom). To solve this problem quickly, the team creates *Technical Debt* (fix), which leads to a decrease in *Late Schedule* and, in the short term, reduces the problem. However, over time, *Technical Debt* accumulates and causes *Rework* (unintended consequence), which contributes to increase the *Late Schedule* again.

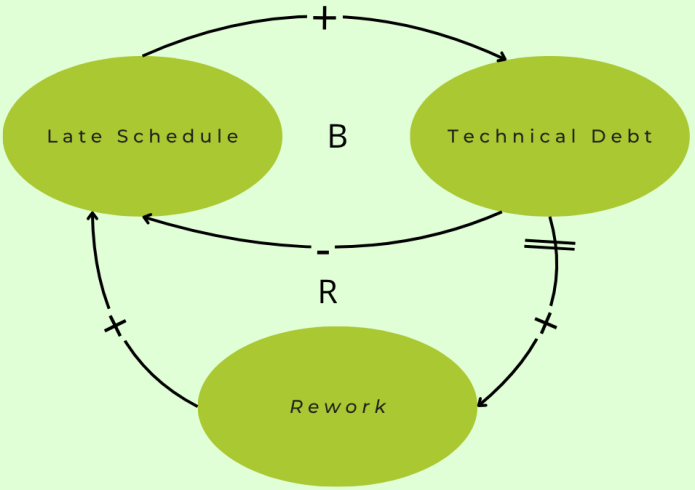

Figure 14. Fix that Fail applied to Software Engineering.

Shifting the Burden

The **Shifting the Burden** archetype occurs when a problem is addressed using a quick, symptomatic solution that temporarily reduces the issue. This short-term fix forms a balancing loop and gives immediate relief. However, because the symptom improves, the need to apply a more fundamental, long-term solution is postponed.

Over time, the repeated use of the quick fix creates side effects that weaken the ability to address the root cause. As a result, the problem keeps returning, the team relies more on the symptomatic solution, and the fundamental solution becomes harder to implement.

Structurally, as shown in Figure 15, the Shifting the Burden archetype includes two balancing feedback loops (B1 and B2) and one reinforcing feedback loop (R) connected by at least four variables.

One balancing loop (B1) represents the symptomatic solution that quickly reduces the problem. The second balancing loop (B2) represents the fundamental solution, which takes longer to produce results and therefore

includes a delay. The reinforcing loop (R) captures the side effects of relying on the quick fix, which gradually reduce the ability to apply the fundamental solution and increase dependence on the symptomatic approach.

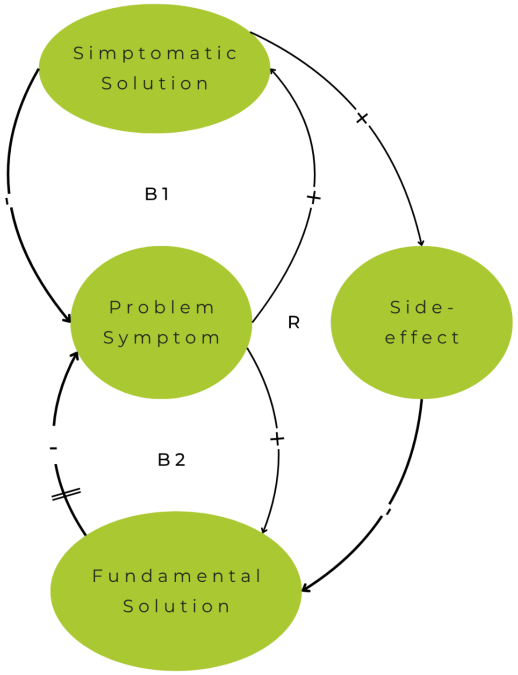

Figure 15. Shifting the Burden modeling pattern.

Figure 16 presents an example of the occurrence of the Shifting the Burden archetype.

Consider that a software team has faced many failures in the system. When *System Failures* (problem symptom) increase, the team applies *Quick Fixes* (symptomatic solution) to solve the problem quickly, and the pressure to adopt *Software Quality Best Practices* (fundamental solution) also increases. This reduces failures in the short term. However, the fundamental solution involves investing in *Software Quality Best Practices*, which may take time to show results. Because *Quick Fixes* provide immediate relief, the adoption of *Software Quality Best Practices* is postponed. Over time, frequent *Quick Fixes* increase *Technical Debt* (side effect), which makes applying *Software Quality Best Practices* more difficult.

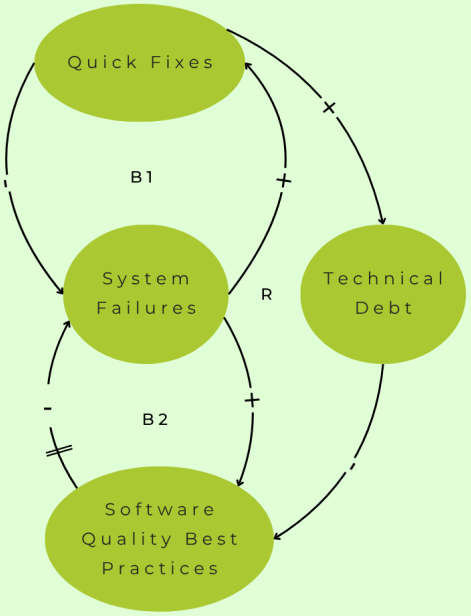

Figure 16. Shifting the Burden applied to Software Engineering.

Limits to Success

The **Limits to Success** archetype describes situations in which an initiative initially produces positive results and sustained growth, creating the perception that success can continue indefinitely. In the early stages, the applied efforts seem sufficient to maintain progress, and the outcomes reinforce the continuation of the same actions. However, as the system evolves, constraints that were not noticeable at first begin to emerge. These limitations gradually reduce the effectiveness of the efforts, slowing growth and establishing a ceiling for performance. Such limits may be associated with internal factors, such as resource capacity or coordination issues, or external factors, such as market saturation and regulations. Over time, these constraints tend to restrain or even reverse the initial gains, characterizing the typical dynamics of this archetype.

The modeling structure of this archetype is presented in Figure 17 and is composed of two main feedback loops: a reinforcing loop (R) that drives the initial growth, and a balancing loop (B) that represents the limiting forces that emerge as performance increases. At first, the reinforcing loop promotes continued investment and improvement, leading to progressively better results. As performance grows, however, constraints begin to appear and activate the balancing loop, which introduces pressures that counteract the growth. Over

time, this balancing dynamic reduces the effectiveness of the reinforcing process, slowing the growth and ultimately establishing a limit to success.

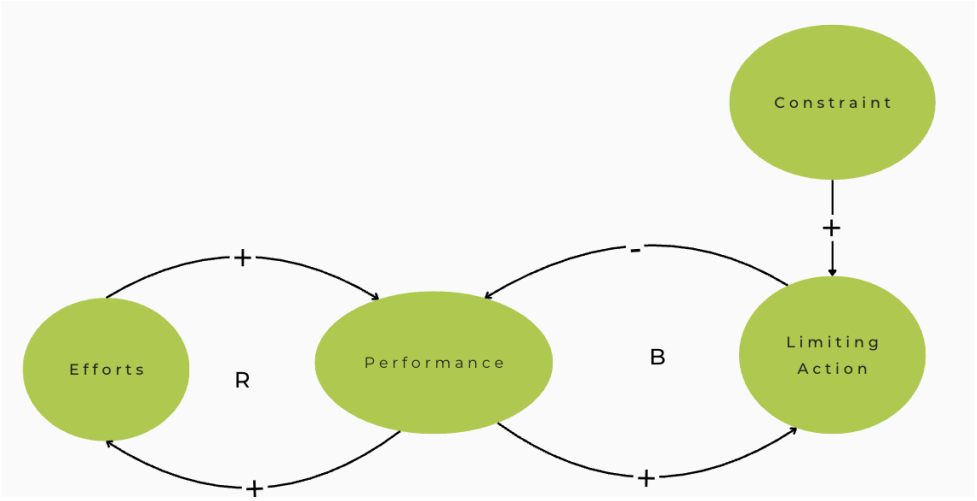

Figure 17. Limits to Success modeling pattern.

Figure 18 presents an example of the occurrence of the Limits to Success archetype.

Consider that a software company has a small team and a growing demand. The company performs *New Hires* (effort) to increase *Delivered Features* (performance). On the other side, increasing *Delivered Features* causes the need to increase *New Hires*, forming a reinforcing loop of progress. However, as *Delivered Features* increase, there is an increase in the need for *Maintenance Activities* (limiting action), which, in turn, decreases *Delivered Features*, because the team needs to spend effort on maintenance activities instead of developing new features. Therefore, *Maintenance Activities*, which are constrained by the *Maintenance Capacity* (constraint), limit the success of the *New Hires* initiative.

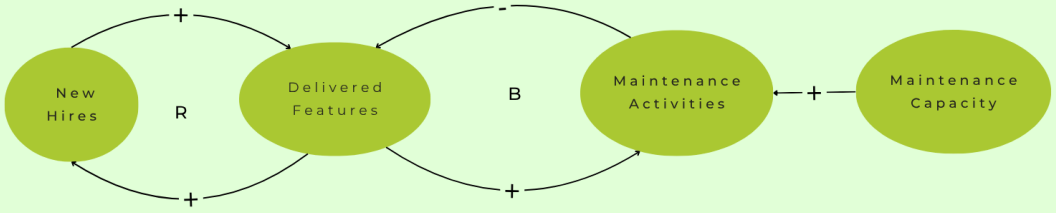

Figure 18. Example of the application of the Limits to Success Archetype.

Drifting Goals

The **Drifting Goals** archetype describes a dynamic where goals that were originally clear and ambitious are gradually lowered in response to pressure or perceived difficulties during the process. Instead of addressing the challenges directly, the system tends to reset its own expectations, creating a cycle where performance falls short of its true potential. This can be compared to adjusting a thermostat: during a harsh winter, if it is hard to keep the house warm, someone might simply lower the desired temperature to avoid overworking the heating system. Similarly, personal or organizational goals are "lowered" to make them seem more achievable, which ultimately compromises quality and excellence.

The modeling structure of the Drifting Goals archetype, illustrated in Figure 19, involves two interconnected balancing feedback loops: the *corrective action* loop (B1) and the *lower goal* loop (B2). When there is a gap between current performance and the desired goal, the *c*orrective action loop attempts to close that gap by implementing improvements. However, due to natural delays, the results often take time to appear. In many situations, the *l*ower goal loop ends up dominating, causing the goal to be adjusted downward, which artificially reduces the gap without addressing the root causes of the problem.

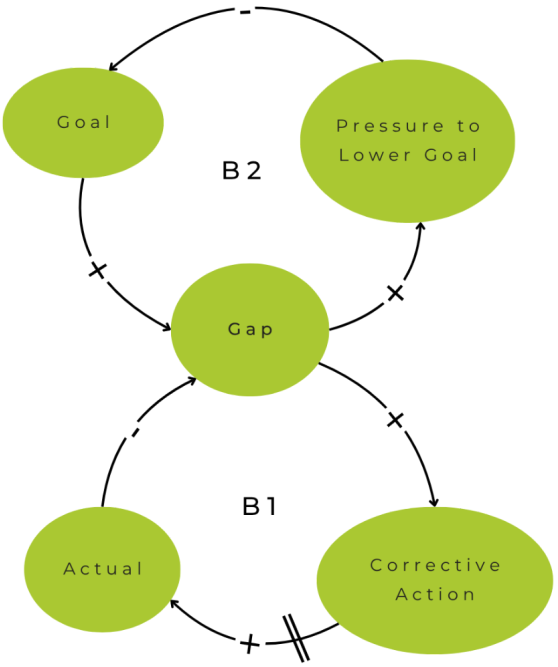

Figure 19. Drifting Goals modeling pattern.

Figure 20 presents an example of the occurrence of the Drifting Goals archetype.

Consider that a software company has established a *Code Quality Goal* (e.g., a target defect density). The difference between the *Code Quality Goal* and the *Actual Code Quality* causes a *Code Quality Gap.* As the *Code Quality Goal* increases, the *Code Quality Gap* increases, while increases in *Actual Code Quality* decrease the *Code Quality Gap.*

As the *Code Quality Gap* increases, *Code Quality Practices* (such as tests, refactoring, and bug fixing) also increase. Increased *Code Quality Practices* improve *Actual Code Quality,* which in turn reduces the *Code Quality Gap* over time.

However, as the *Code Quality Gap* increases and persists, the *Pressure to Lower Code Quality Goal* increases. As the *Pressure to Lower Code Quality Goal* increases, the *Code Quality Goal* decreases. A decrease in the *Code Quality Goal* directly decreases the *Code Quality Gap*, without improving the *Actual Code Quality*.

Over time, repeated increases in the *Pressure to Lower Code Quality Goal* lead to repeated decreases in the *Code Quality Goal,* causing the expected code quality standard to drift downward.

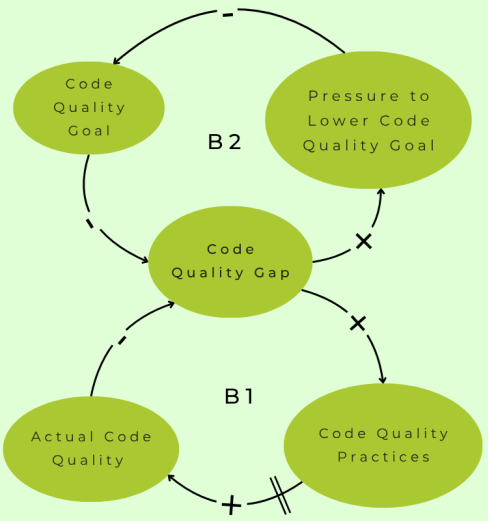

Figure 20. Illustration of the Drifting Goals in a Software Engineering context.

Growth and Underinvestment

The **Growth and Underinvestment** archetype describes situations in which a system has the potential to grow, but this growth is limited because investments in essential resources are insufficient. At first, the system performs well, creating the expectation that growth will continue. However, as demand increases, the available capacity is not expanded at the same pace. As a result, performance begins to deteriorate, and the system fails to reach its full potential, leading to stagnation or even decline .

The structure of this archetype, illustrated in Figure 21, extends the Limits to Success archetype by adding a third balancing loop. The reinforcing feedback loop (R) drives growth, increasing performance over time. As performance grows, limiting factors such as resource constraints activate the balancing feedback loop (B1), which slows down progress. To address this limitation, a second balancing feedback loop (B2) represents investments in capacity intended to improve performance and support continued growth. However, delays in recognizing the need for investment, making decisions, and implementing improvements reduce the effectiveness of this loop. Because of these delays, capacity is expanded too late, allowing the limiting factors to dominate and restricting the system's growth.

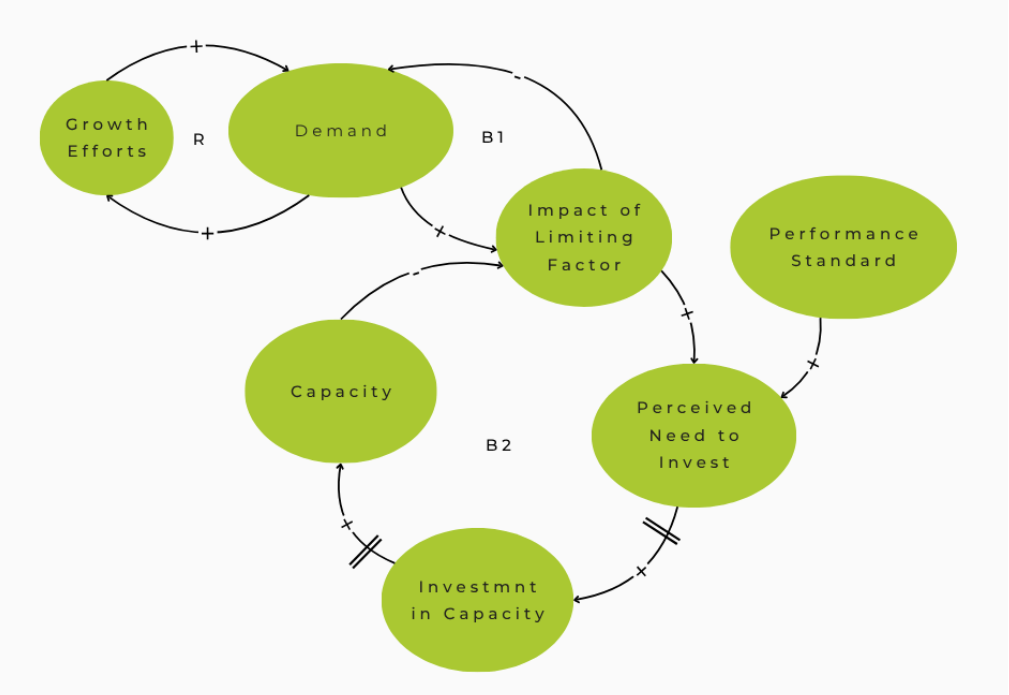

Figure 21. Growth and Underinvestment modeling pattern.

Figure 22 presents an example of the occurrence of the Growth and Underinvestment archetype.

Consider a software company experiencing rapid demand for new features. Initially, an increase in *Feature Demand* (demand) stimulates higher *Delivery Speed* (growth effort), as the development team accelerates feature implementation to respond to market needs. This creates a reinforcing dynamic in which greater *Delivery Speed* helps sustain growing *Feature Demand*.

However, as *Feature Demand* increases, the *Defect Rate* (Impact of limiting factors) also rises due to reduced testing and growing system complexity. A higher *Defect Rate* negatively impacts *Feature Demand*, since defects slow releases and reduce the ability to deliver new functionality. At the same time, the increase in *Defect Rate* widens the *Delivery Gap* (performance standard), which represents the difference between the *Desired Delivery Speed* (perceived need to invest) and the current delivery performance. As the *Delivery Gap* grows, it increases the need for *Investment in Test Automation* (investment in capacity).

As the *Investment in Test Automation* increases, *Testing Capacity* increases over time. Increased *Testing Capacity* reduces the *Defect Rate* by enabling more effective and earlier detection of defects.

Lower *Defect Rate* allows *Delivery Speed* to recover and supports the system's ability to meet *Feature Demand.* However, when investment in *Testing Capacity* (capacity) is delayed or insufficient, the *Defect Rate* remains high, limiting delivery performance and preventing the system from sustaining its initial growth.

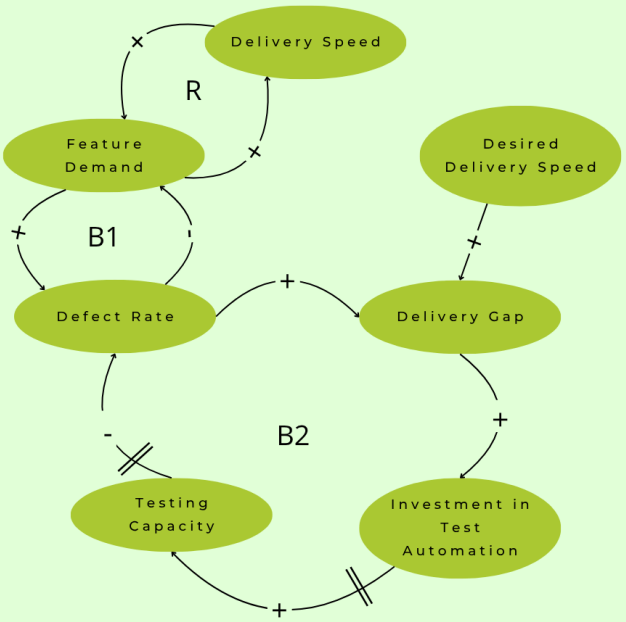

Figure 22. Example scenario represented by the Growth and Underinvestment archetype.

Success to the Successful

The **Success to the Successful** archetype describes a dynamic in which resources are distributed unevenly between two or more competing parties, favoring those that initially achieve greater success. This mechanism reinforces the competitive advantage of the more successful group, making it harder for others to catch up. As a result, the disparity between the parties increases over time and may eventually lead to an excessive concentration of opportunities in a single agent or group.

The structure of this archetype, illustrated in Figure 23, is based on two interconnected reinforcing feedback loops (R1 and R2) linked through a central variable: the preferential allocation of resources. When one agent (A) begins to receive more resources, its success increases, reinforcing the decision to keep investing in it (R1). At the same time, this preference reduces the resources available to the competing agent (B), limiting its growth and making it increasingly less competitive (R2). This creates a compounding effect where the initial advantage tends to perpetuate itself, consolidating A's dominance while undermining B.

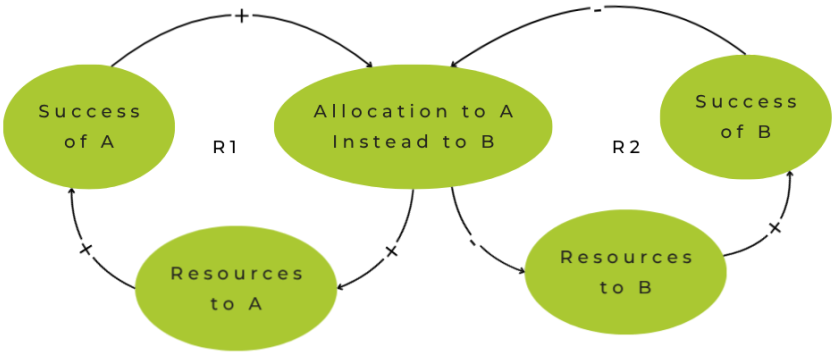

Figure 23. Success to the Successful modeling pattern.

Figure 24 presents an example of the occurrence of the Success to the Successful archetype.

Consider a software company with two teams competing for the same workforce: a feature development team and a test automation team. Initially, both teams receive similar *Resources to the Feature Team* and *Resources to the Test Team*, allowing them to achieve comparable results. Over time, the *Success of the Feature Team* begins to increase, since delivering new features generates visible business value. As a result, management increases the *Resource Allocation to Feature Team instead of Test Team*, assigning more developers to feature development activities. This decision increases the *Resources to the Feature Team*, which further improves the *Success of the Feature Team*.

However, because the same workforce must be shared, increasing the *Allocation to Feature Team instead of Test Team* reduces the *Resources to the Test Team*. With fewer resources available, the *Success of the Test Team* declines, for example due to reduced test automation, lower coverage, and more regression defects. As the *Success of the Test Team* decreases, management perceives less benefit in investing in testing and further increases the *Resource Allocation to the Feature Team instead of Test Team*. Consequently, the *Resources to the Test Team* continue to shrink, reinforcing the advantage of the feature team.

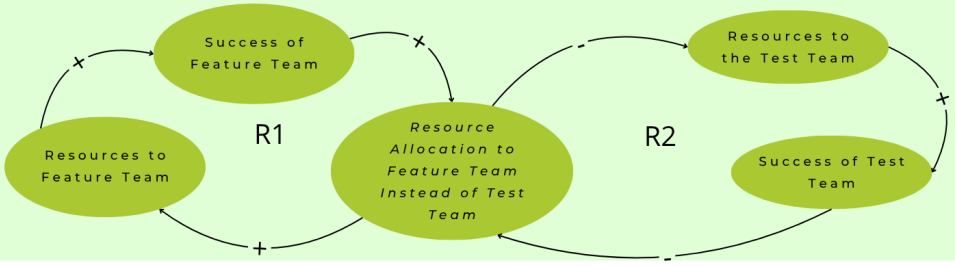

Figure 24. Application of the Success to the Successful archetype.

Escalation

The Escalation archetype describes a competitive dynamic in which two or more parties respond to each other's gains by increasing their own efforts, leading to a cycle of continuous intensification. This behavior can generate positive outcomes, such as innovation and improved efficiency, but it may also produce harmful effects, including prolonged conflict, wasted resources, and organizational burnout .

The modeling structure of this archetype is illustrated in Figure 25. It involves two interconnected balancing loops (B1 and B2). Each party aims to improve its relative position in response to the perceived threat posed by the other's performance. When agent A increases its efforts, its results improve, strengthening its competitive position (B1). However, this improvement causes agent B to feel threatened, prompting it to intensify its own actions to regain lost ground (B2). This process repeats in a self-reinforcing pattern, resulting in a progressive escalation of actions on both sides .

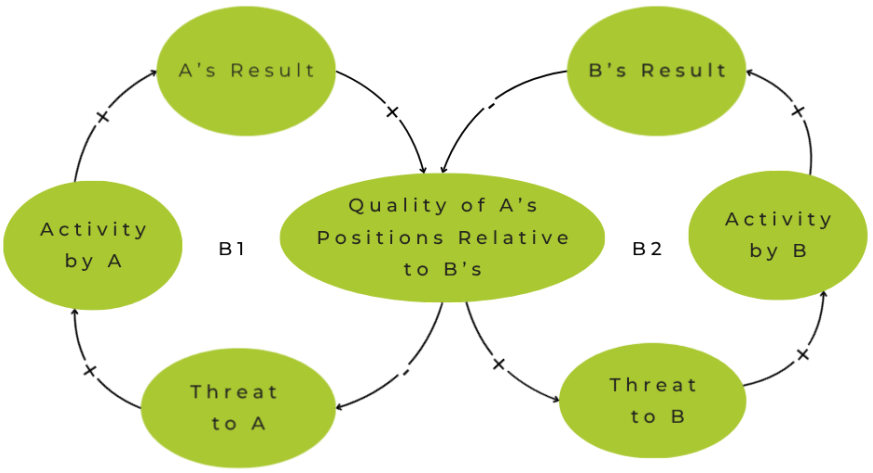

Figure 25. Escalation Architecture modeling pattern.

Figure 26 presents an example of the occurrence of the archetype.

Consider that two software companies (A and B) compete for qualified developers, as these resources determine their delivery capacity. When *Company A's delivery capacity* (A's result) increases, due to hiring more developers, this improves *A's delivery capacity relative to B* (quality of A position relative to B). This improvement is perceived by Company B as a greater *competitive threat to B* (threat to B), leading it to intensify its *investment in hiring developers* by B (activity by B). With more developers, *Company B increases its delivery capacity* (B's result), reducing *Company A's relative position compared to B* (quality of A's position relative to B). This reduction is perceived by Company A as a greater *competitive threat to A* (threat to A), prompting it to increase its *investment in hiring developers* by A(activity by A), thereby raising its delivery capacity again.

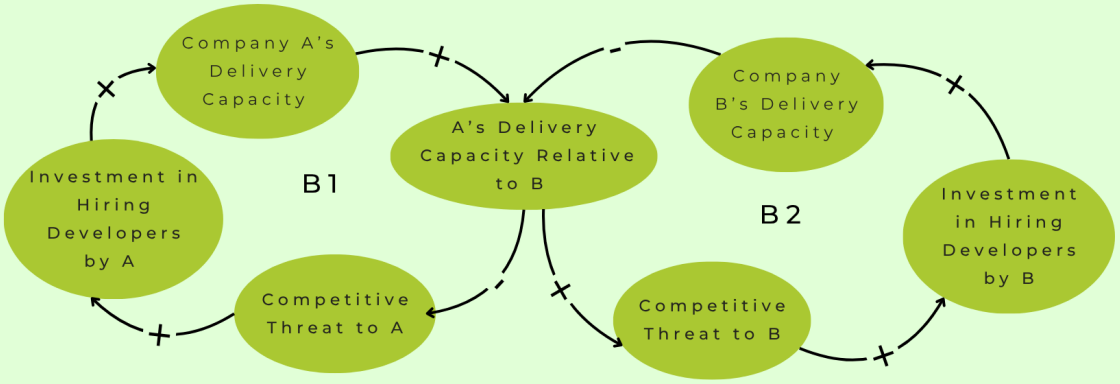

Figure 26. Example of Escalation archetype in a Software Engineering context.

Tragedy of the Commons

The Tragedy of the Commons archetype describes a dynamic in which multiple agents exploit a shared resource to maximize their benefits. However, as each agent increases their usage, the resource's capacity deteriorates, ultimately harming everyone involved. This phenomenon can be observed in environments where excessive use of a common resource can lead to systemic failure .

The modeling structure of this archetype, illustrated in Figure 27, involves multiple agents using a shared resource to obtain personal gains. Initially, each agent perceives that increasing their activity leads to direct benefits, without accounting for the cumulative impact on the system. This process is driven by two reinforcing feedback loops (R1 and R2), where each agent's activity produces increasing net gains. Additionally, other two reinforcing feedback loops (R3 and R4) represent the delay in perceiving the total impact of that activity, showing how individual contributions are not immediately understood in the broader system context. However, as the demand for the resource grows, the limitations of that resource become more visible. The relationship between total activity and the declining benefit per activity introduces two balancing feedback loops (B1 and B2), which act as late-stage regulatory mechanisms.

Due to this delay, agents continue to increase their usage even as the benefit per action declines. The result is progressive depletion of the shared resource, leading to worsening outcomes for all involved .

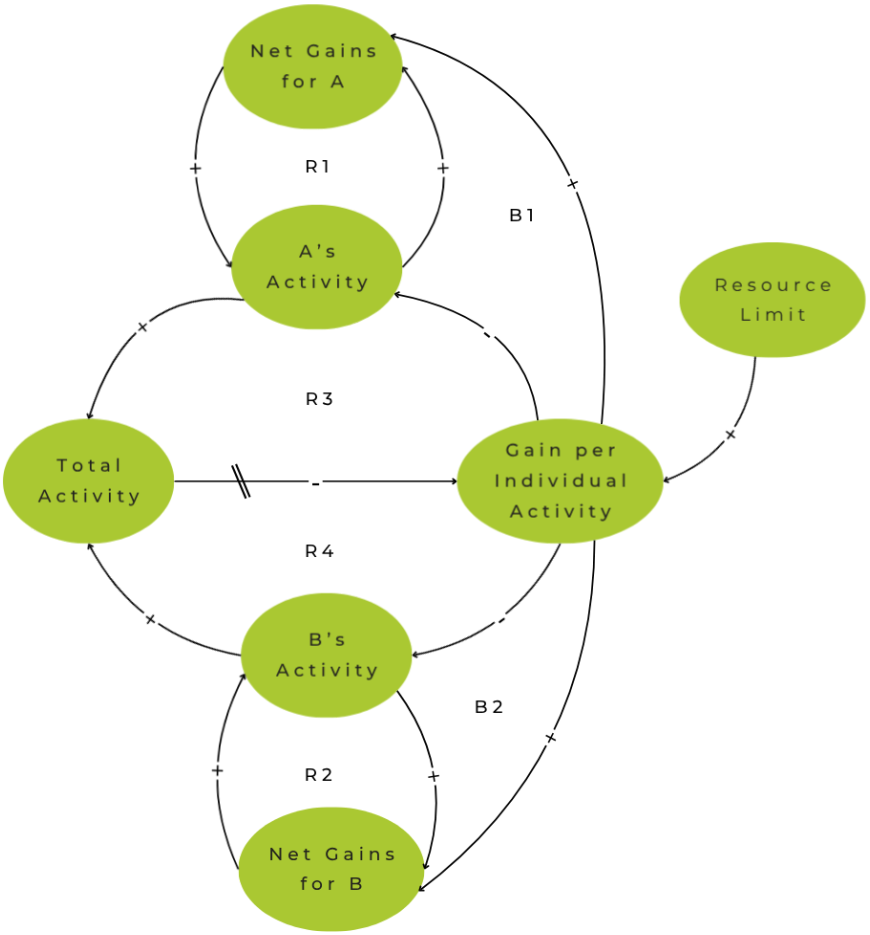

Figure 27. Tragedy of the Commons modeling pattern.

Figure 28 presents an example of the occurrence of the Tragedy of the Commons archetype.

Consider that in a modern cloud-native environment, multiple autonomous service teams (A and B) develop microservices that rely on a shared infrastructure with limited capacity. Each team seeks to maximize its own outcomes by increasing its *Resource Allocation Request* (A's activity / B's activity), since higher resource availability improves *Service Performance* (Net Gains for A / Net Gains for B). Locally, this behavior is rational, as increased allocation leads to better performance, reinforcing further requests for resources.

However, as both teams expand their requests, the *Total Cluster Load* (Total Activity) increases. Given the existence of a *Shared Infrastructure Capacity Limit* (Resource Limit), the system becomes progressively constrained. As a result, the *System Stability* (Gain per Individual Activity) begins to degrade, reducing the effectiveness of each unit of allocated resource. Due to a delay in perceiving this degradation, teams continue to increase their allocation requests, further intensifying the load on the system.

Over time, the reduced system stability negatively impacts the *Service Performance of teams A* and Service Performance of teams B, undermining the very gains each team sought to achieve.

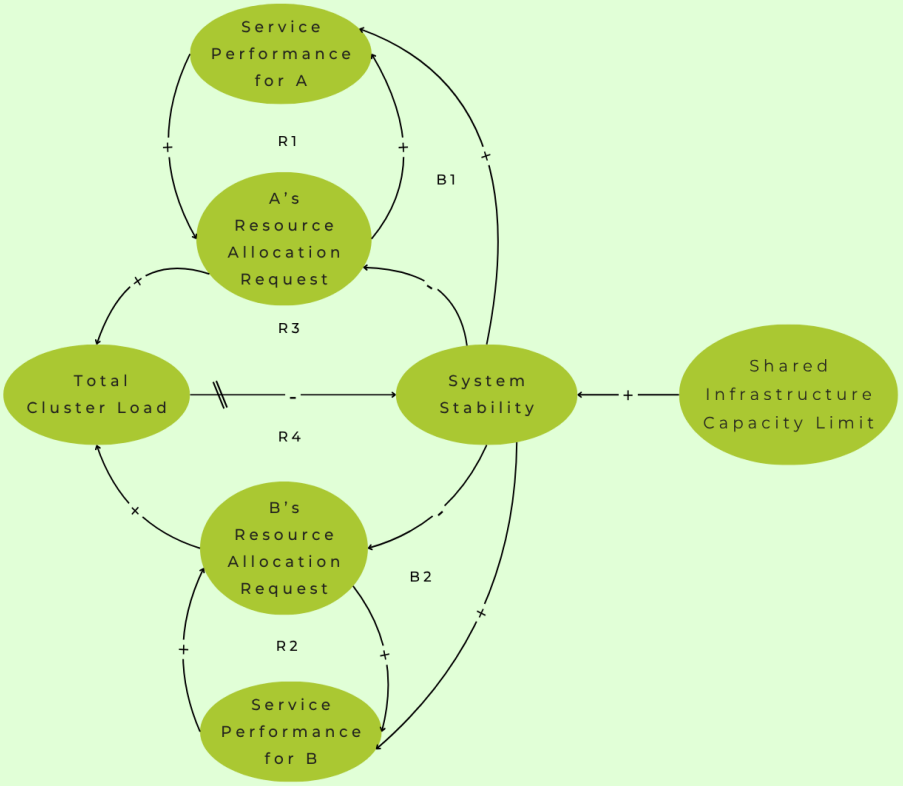

Figure 28. Tragedy of the Commons archetype example in Software Engineering.

PART II: PRACTICAL GUIDANCE

2. BUILDING A CLD STEP BY STEP

In Section 1, we provided basic knowledge about System Thinking and Causal Loop Diagram. Now, in this section, we offer guidance on how to create a CLD. First, we briefly describe an illustrative scenario and, subsequently, we provide a step-by-step process to build a CLD and we use the illustrative scenario to demonstrate it.

2.1 ILLUSTRATIVE APPLICATION SCENARIO[1]

A Brazilian organization, referred here as Organization A, collaborates with a European partner, Organization B, to deliver software products to clients abroad. Organization B elicits requirements from clients, and Organization A is in charge of developing the corresponding software. Because of the increasing number of projects and team members, added to the lack of flexible processes, some problems emerged, such as projects being late and over budget, an increase in software defects, overloading of the teams due to rework on software artifacts, and communication issues among client, Organization A, and Organization B. Recurring misunderstandings regarding requirements and priorities began to intensify coordination difficulties between the organizations. Moreover, there has been a growing pressure for faster deliveries, often at the expense of software quality.

2.2 STEP BY STEP

Figure 29 illustrates the main e steps you should follow to build a CLD.

[1] This scenario is described in (DOS SANTOS JÚNIOR et al., 2022). In that study, CDLs were applied to support the implementation of agile practices in a software organization.

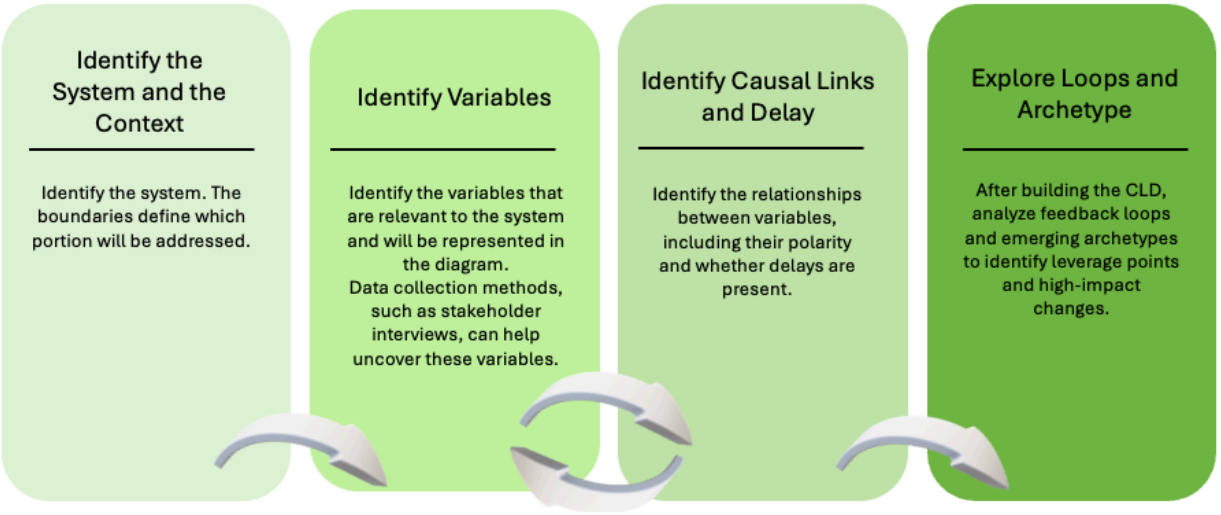

Figure 29. Steps to build a CLD

The first step consists in defining the context to which the CLD will be developed, i.e., which system will be represented. Once the context is established, the relevant variables involved in the system must be identified. When identifying the variables, the modeler can also identify their causal relationships with other variables, including the polarity of the links and delays when applicable. Thus, variables, causal links, and delays are defined iteratively as the modeler progressively builds and refines the CLD. As the CLD is built, feedback loops and archetypes can be identified, helping the modeler interpret the system behavior and identify improvement actions.

It is worth noticing that, although we present the steps sequentially (for sake of clarity and simplicity), in practice, the process can be iterative. For example, after identifying feedback loops, the modeler can realize that new variables and relationships should be added to do CLD and go back to previous steps to add them. Next, we provide further information about the steps and exemplify them for the illustrative scenario described in Section 2.1.

Step 1: Identify the System and the Context

The first step is to define the system and the context to which the CLD will be applied. This system may be, for example, an organization, a team, or a specific domain. Establishing the boundaries helps delimit the context to be considered and is essential to ensuring an effective and representative modeling of the studied reality. Clearly defining the system boundary helps establish the scope of analysis and identify which variables and relationships are relevant to the system and which ones lie outside the current focus. These boundaries are not fixed; they can be expanded or narrowed as understanding evolves. Defining

objectives, mapping stakeholders, and discussing different perspectives further clarify and refine the system boundaries.

Getting to know the context to be considered helps establish the boundaries that delimit the system and provide a panorama of the scenario to be modeled.

As an example, consider the following context, established to the illustrative scenario presented in Section 2.1.

The system involves the software development area of two organizations: a Brazilian software organization (Organization A) that collaborates with a European organization (Organization B) to deliver software. Organization B elicits requirements from clients, and Organization A is in charge of developing the corresponding software.

The context to be considered (i.e., the boundaries that delimit the system to be modeled) regards the interaction between Organization A and B teams when developing software and the effects of this interaction on the projects. It can be summarized as follows (Figure 30):

Organization B often produces requirements poorly specified, neither adopting a proper technique nor following a pattern to describe them. Thus, the development teams of Organization A frequently misunderstand the requirements that describe the software, component, or functionality to be developed. Misunderstood requirements contribute to increasing the number of defects in the produced software artifacts, since design, code, and test are produced based on the requirements informed by Organization B. To fix defects, Organization A mobilizes (and often overloads) the development team that needs to perform new urgent development activities. These urgent (fixing) activities increase the number of defects fixed through rework and temporarily decrease the number of defects in the produced software artifacts. These activities are performed as fast as possible, aiming not to delay other activities. Thus, they do not properly follow software quality best practices. Moreover, they contribute to increasing the project cost and time, leading to late and over-budget projects. In parallel, the growth of defects in software artifacts also increases the pressure to adopt software quality techniques, which, when systematically applied, tend to reduce defect recurrence in a more sustainable way.

Figure 30. Context considered in the illustrative scenario.

Step 2: Identify Variables

After defining the context, the modeler should identify the variables to be included in the CLD. These variables must be relevant to the chosen context, reflecting key factors that influence the system's behavior. The selection of

variables ensures that the model accurately represents the dynamics of the system under analysis.

Data collection methods, such as interviews with stakeholders, can be employed to identify these variables and understand their relationships with others. Conducting structured or semi-structured interviews with domain experts, team members, and decision-makers helps capture different perspectives on the system's behavior. This approach ensures that the selected variables are not only theoretically relevant but also aligned with the practical challenges and dynamics of the real-world context.

To exemplify, next we present some variables identified in the context defined in the previous step. They are highlighted in ***bold italics*** in the text below.

Organization B often produces ***requirements poorly specified***, neither adopting a proper technique nor following a pattern to describe them. Thus, the development teams of Organization A frequently misunderstand the requirements that describe the software, component, or functionality to be developed. ***Misunderstood requirements*** contribute to increasing the number of ***defects in the produced software artifacts***, since design, code, and test are produced based on the requirements informed by Organization B. To fix defects, Organization A mobilizes (and often overloads) the development team that needs to perform ***new urgent development activities***. These urgent (fixing) activities increase the number of ***defects fixed through rework*** and temporarily decrease the number of defects in the produced software artifacts. These activities are performed as fast as possible, aiming not to delay other activities. Thus, they do not properly follow software quality best practices. Moreover, they contribute to increasing the project cost and time, leading to ***late and over-budget project***s. In parallel, the growth of defects in software artifacts also increases the pressure to adopt ***software quality techniques***, which, when systematically applied, tend to reduce defect recurrence in a more sustainable way.

Figure 31. Identification of variables.

Figure 32 represents the variable identified in the Organization A-B scenario.

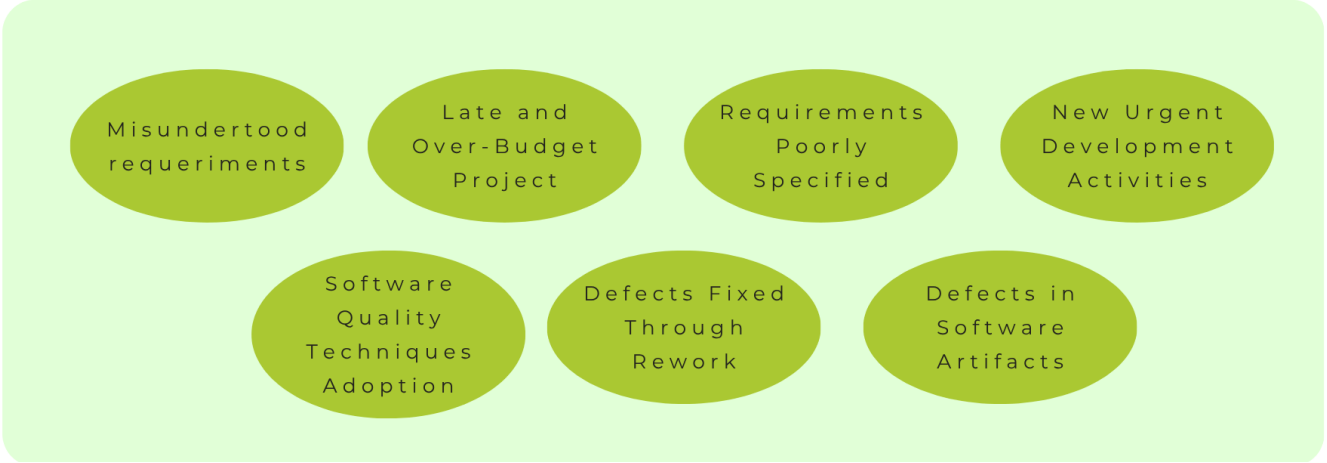

Figure 32. Variables identified in the illustrative scenario.

Step 3: Identify Causal Links and Delay

Once variables are identified, it is necessary to identify the relations – i.e., the causal links – between them. As stated earlier, this step and step 2 (Identify Variables) can be performed interactively. In this way, when identifying variables, the modeler can also identify the relations between them and repeat these steps until all variables and relations are identified.

These links define how changes in one variable influence others, shaping the system's behavior over time.

When identifying a causal link, the modeler must define its polarity (positive or negative), to properly represent how a change in the source variable affects the target one. Moreover, the modeler must indicate when delays happen, indicating that the effects of changes in the source variable may take time to be perceived in the target variable.

The result produced after performing steps 1 and 2 is a CLD composed of variables and links.

By considering the context and variables previously defined, the following relationships can be identified:

Positive causal links (i.e., a change in the source variable changes the target variable in the same way) is illustrated in Figure 33:

(↑/↓) Requirements Poorly Specified cause (↑/↓) ***Misunderstood Requirements***

(↑/↓) ***Misunderstood Requirements cause*** (↑/↓) ***Defects in Software Artifacts***

(↑/↓) ***Defects in Software Artifacts*** cause (↑/↓) ***New Urgent Development Activities***

(↑/↓) ***New Urgent Development Activities*** cause (↑/↓) ***Late and Over-budget Project***

(↑/↓) ***Late and Over-budget Project*** cause (↑/↓) ***Defects Fixed Through Rework***

(↑/↓) ***Defects in Software Artifacts cause (a need for)*** (↑/↓) ***Software Quality Techniques Adoption***

Figure 33. Positive causal links identified in the context.

Negative causal links (i.e., a change in the source variable changes the target variable in the opposite way) is illustrated in Figure 34:

(↑/↓) New Urgent Development Activities *cause* ***(↓/↑) Defects in Software Artifacts***

(↑/↓) Defects Fixed Through Rework *cause (a need for)* ***(↓/↑) Software Quality Techniques***

(↑/↓) Software Quality Techniques *cause* ***(↓/↑) Defects in Software Artifacts.*** *In this relation, the effects of applying Software Quality Techniques take time to be perceived in the Defects in Software Artifacts, meaning that there is a delay.*

Figure 34. Negative causal links identified in the context.

Figure 35 illustrates the CLD representing these variables and relationships.

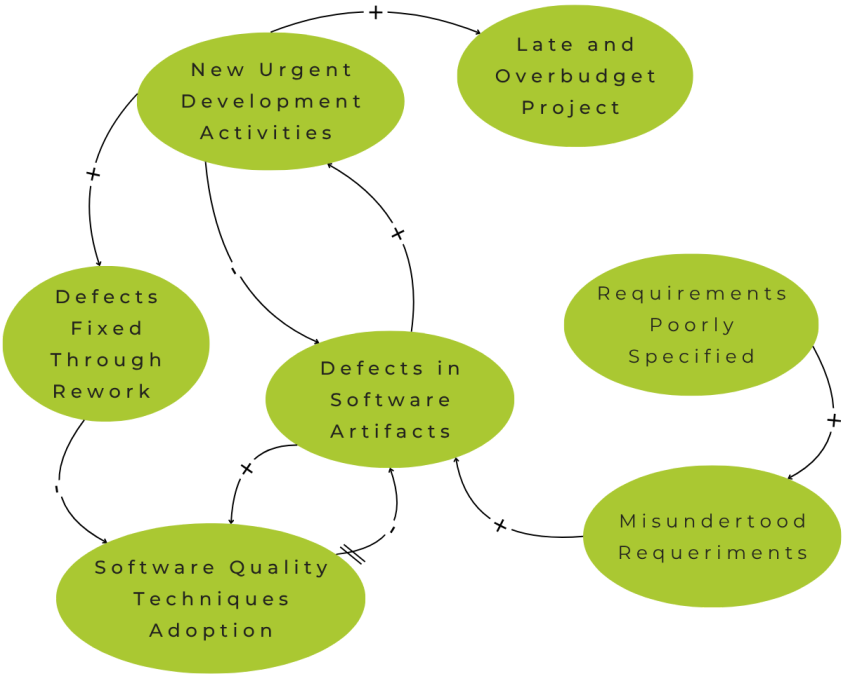

Figure 35. CLD composed of the identified variables, causal links and delays

Step 4: Explore Loops and Archetypes

Once variables and relationships are established, the CLD is produced and can be analyzed. At this stage, the modeler can examine the diagram to identify feedback loops, determine whether they are reinforcing or balancing, and analyze how these loops drive the system's behavior over time. In addition, the modeler can investigate the occurrence of archetypes by recognizing recurring patterns formed by the interactions among variables.

Feedback loops and archetypes reveal underlying structures and potential unintended consequences within the system and help obtain a deeper understanding of the system's overall structure.

By analyzing archetypes and feedback loops, it becomes possible to identify leverage points and opportunities for improvement in the defined context. By understanding how specific structures generate problematic behaviors, the modeler can propose targeted interventions that modify key relationships, adjust delays, or introduce new variables. These actions support more effective decision-making and enable improvements focused on changing the system's structure rather than only addressing its symptoms.

By analyzing the CDL illustrated in Figure 35, it is possible to identify two balancing feedback loops (B1 and B2) and one reinforcing feedback loop (R) organized in a modeling pattern that reveals an occurrence of the *Shifting the Burden* archetype. Figure 36 shows the CLD produced in the previous step, now identifying its feedback loops (B1, B2, and R) and archetype (highlighted in blue).

In the analyzed scenario, ***Requirements Poorly Specified*** increase ***Misunderstood Requirements***, which lead to an *increase* in ***Defects in Software Artifacts***. To deal with these defects, the organization often resorts to urgent corrective efforts by overloading the development team with ***New Urgent Development Activities***, which temporarily reduces the defects. However, these quick fixes typically bypass software quality best practices. In parallel, the presence of defects also causes a need for increasing ***Software Quality Techniques Adoption***, which, when applied, helps reduce defect recurrence more sustainably. The *Shifting the Burden* archetype emerges as, to solve a *problem* (***Defects in Software Artifacts***), the organization repeatedly favors a *symptomatic solution* (***Urgent Development Activities***) over the fundamental one (***Software Quality Techniques Adoption***), producing a *side-effect* (***Defects Fixed through Rework***). These dynamics form a reinforcing loop where temporary fixes mask the deeper issue, reducing the perceived need for structural improvements and ultimately perpetuating the cycle of rework and overload.

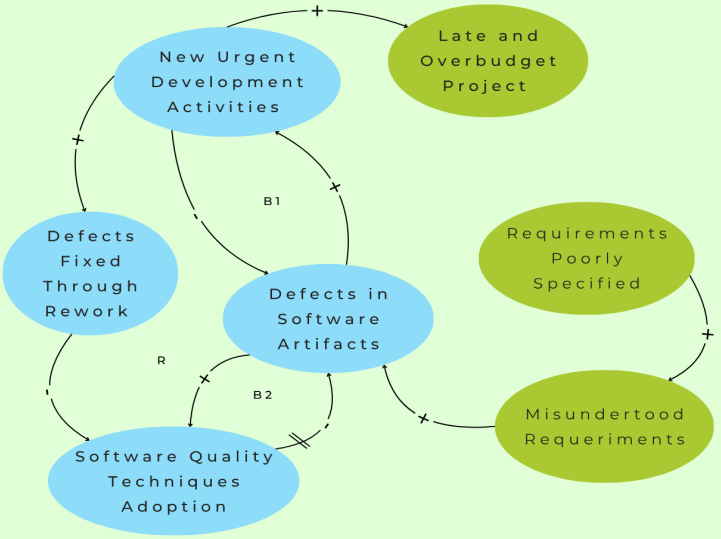

Figure 36. CLD which feedback loops and an occurrence of the Shifting the Burden archetype.

PART III: SUPPORTING TOOL

CaLMo: Causal Loop Diagram Modeler (Apolinário et al., 2025)

In the previous sections, we provided basic knowledge about CLDs and guidelines to help build them. In this section, we present CaLMo, a tool to support creating and analyzing CLDs. The content presented in this section was extracted from (Apolinário et al., 2025) and (Lahass et al., 2026).

3.1 Introducing CaLMo

Several tools can be used to create CLDs. General-purpose diagramming or presentation tools (e.g., Visual Paradigm and draw.io) and generic online diagramming or whiteboard tools (e.g., Miro, Creately) are often used. However, although these tools facilitate drawing CLDs and, in some cases, provide rich visualization and collaboration features, they treat diagrams essentially as graphical artifacts: they do not maintain the underlying CLD structure as first-class data, do not automatically identify feedback loops or archetypes, and do not offer support for checking polarity assignments, modeling rules, or model consistency. As a consequence, the burden of ensuring that CLDs are well-defined, consistent, and analytically useful falls entirely on modelers, making the activity error-prone and time-consuming.

To help address this gap, we developed **CaLMo** (Causal Loop Diagram Modeler (Apolinário et al., 2025). CaLMo supports users in creating and analyzing CLDs by enabling them to define variables and relationships between them, and by automatically identifying feedback loops and archetypes. In contrast to general-purpose diagramming tools that treat CLDs as drawings, CaLMo represents CLDs as structured models (with variables and relationships as first-class data) and leverages this representation to perform consistency checks, prevent common modeling mistakes, and drive automated analyses, including archetype detection. Currently, the tool detects the eight core system archetypes presented in Part I of this document, and provides visualization mechanisms (such as color coding and legends) to help users interpret complex and overlapping systemic structures.

CaLMo is available at https://dev.nemo.inf.ufes.br/calmo/. A short video introducing CaLMo can be found at https://doi.org/10.5281/zenodo.15477387.

3.2 CaLMo Features

To start using CaLMo, the user must first access the tool through its web interface (https://dev.nemo.inf.ufes.br/calmo/). New users are required to create an account to gain full access, enabling the storage and management of their diagrams. Once logged in, the user is directed to the Dashboard homepage (Figure 37), which serves as the central navigation hub, offering immediate access to the core functionalities, which include: manage variables, create CLDs, and visualize CLDs. On this page, the user can also access tutorial resources.

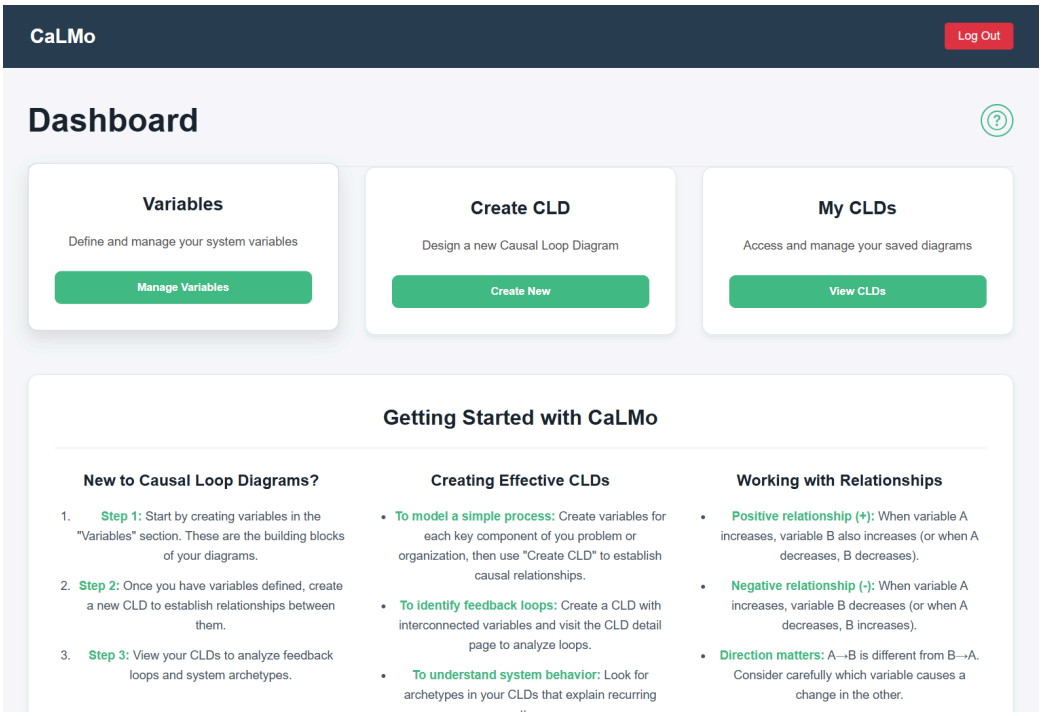

Figure 37. CaLMo homepage.

To create a CLD, the user must first create variables to be included in the CLD. For that, the user clicks the **Manage Variables** button on the homepage and gets to the variables page, where they can record variables that will be later used in CLDs. Each variable must have a name and can have an optional description to enhance the model's clarity and maintainability. Following the scenario described in Section 2.2.2, Figure 38 illustrates some created variables and the addition of the *Software quality techniques* variable. By recording the variables, the user can add them to several CLDs. In this way, the tool reduces rework and helps keep consistency in different diagrams.

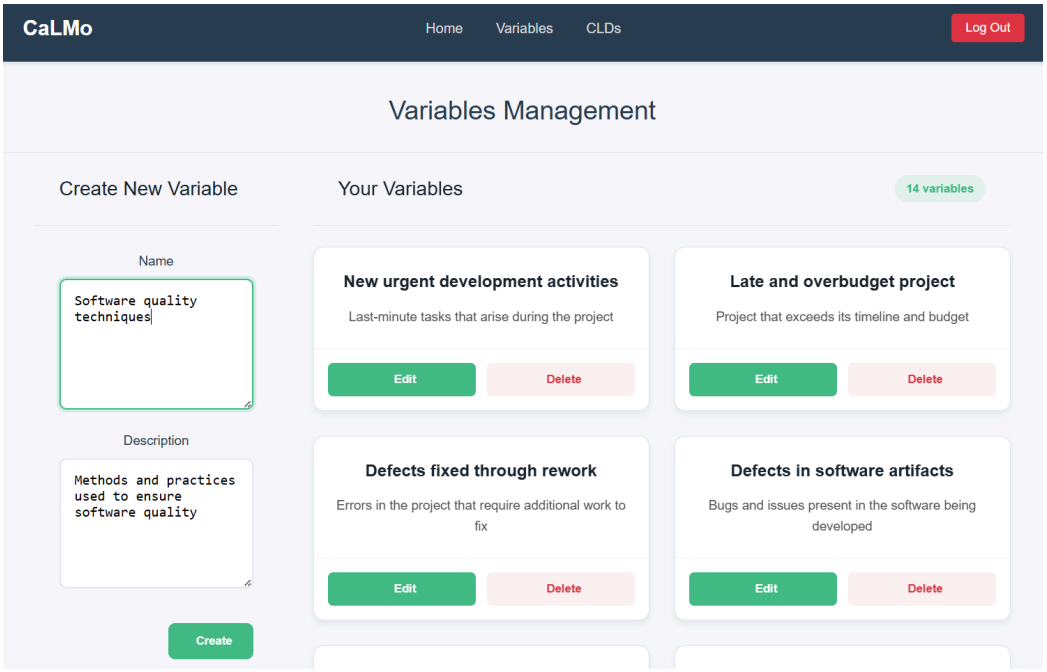

Figure 38. Variable management.

Following the creation of variables, the user can create CLDs using the previously created variables (Figure 39). In the **Create New CLD** page -- accessed from the homepage via the Create New button -- the user must identify the CLD to be created by providing its name and an optional description to help distinguish between multiple CLDs. The tool also records the CLD creation date, editable by the user. The user must, thus, select the variables to be added to the diagram and identify the relationships between them by indicating the source and target variables and the type of relationship (positive or negative). CaLMo checks the consistency of the informed relationships and prevents the user from making mistakes (e.g., contradictory polarities or circular references between variables).

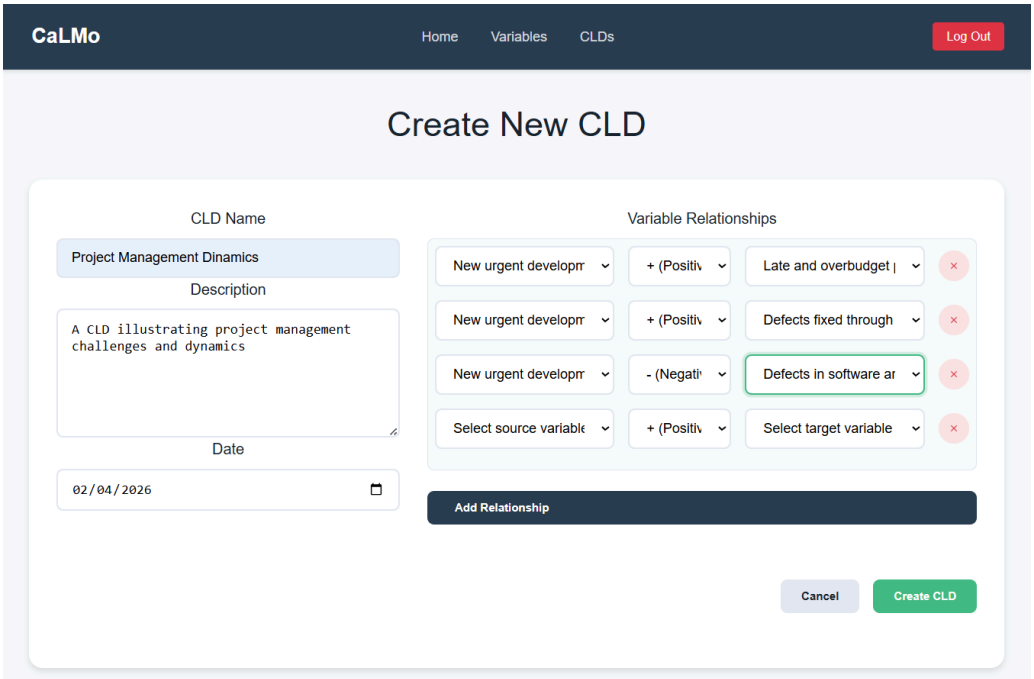

Figure 39. Relationship and CLD creation.

Once the user defines the desired variables and causal relationships, CaLMo creates the corresponding CLD, stores it in the user's CLD library, and displays it to the user. Figure 40 shows the CLD generated for the illustrative scenario presented in Section 2.1. For each detected archetype, the tool assigns a unique node color and indicates it in the *Legend — Archetypes in this CLD* area (below the diagram). If CaLMo detects no archetypes in the CLD, the legend area switches to a warning message to inform the user that no behavior pattern was identified in the CLD. In the example, nodes (variables) that belong to the *Shifting the Burden* archetype are rendered in green, and the legend displays a green marker indicating that. The remaining variables appear in a neutral yellow tone. Positive relationships are represented by green arrows and negative relationships by red arrows, making polarity visually explicit. Users can customize the CLD by repositioning nodes to improve layout clarity.

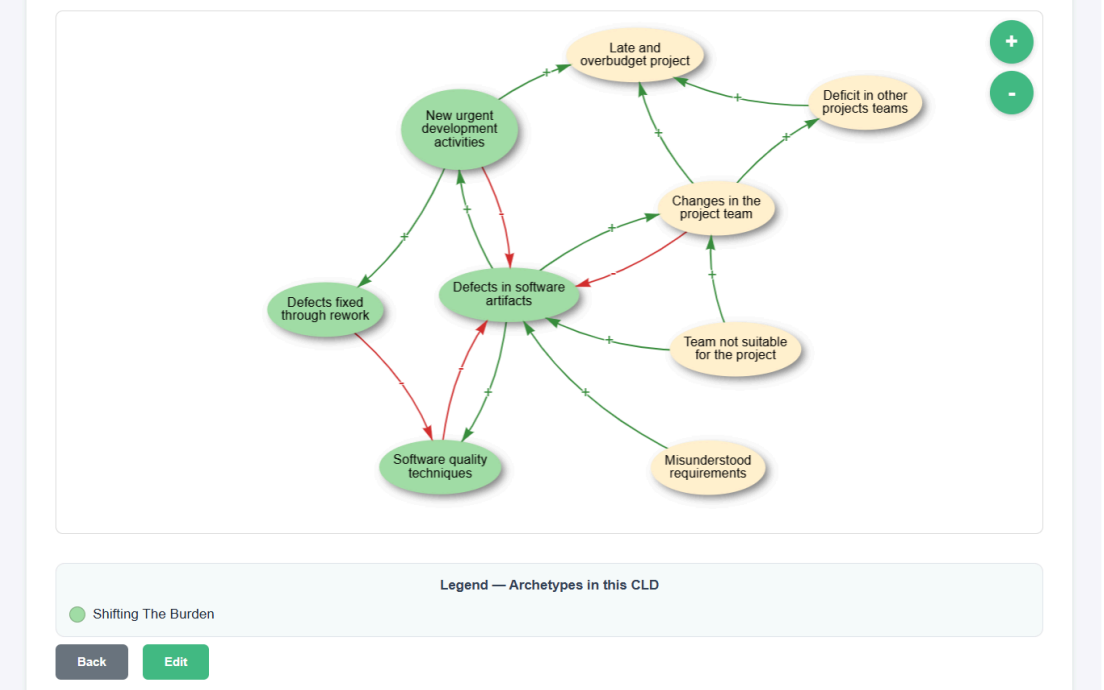

Figure 40. Generated CLD containing the Shifting the Burden archetype.

When a variable is part of more than one archetype, CaLMo applies the color scheme simultaneously (i.e., the variable node is presented with more than one color). Figure 41 illustrates a CLD in which a variable is part of *Shifting the Burden* and *Fixes that Fail* archetypes. Variables that belong only to *Shifting the Burden* are rendered in green (e.g., *New urgent development activities, Defects fixed through rework, Software quality techniques*), whereas those that belong only to *Fixes that Fail* appear in purple (e.g., *Overtime work, Team fatigue*). The variable *Defects in software artifacts* is part of both archetypes and is therefore displayed as a segmented node that combines the two colors, explicitly signaling its central role in overlapping systemic structures. The legend below the diagram shows a green marker for *Shifting the Burden* and a purple marker for *Fixes That Fail*, allowing users to quickly relate each color (and each node segment) to the corresponding archetype instance.

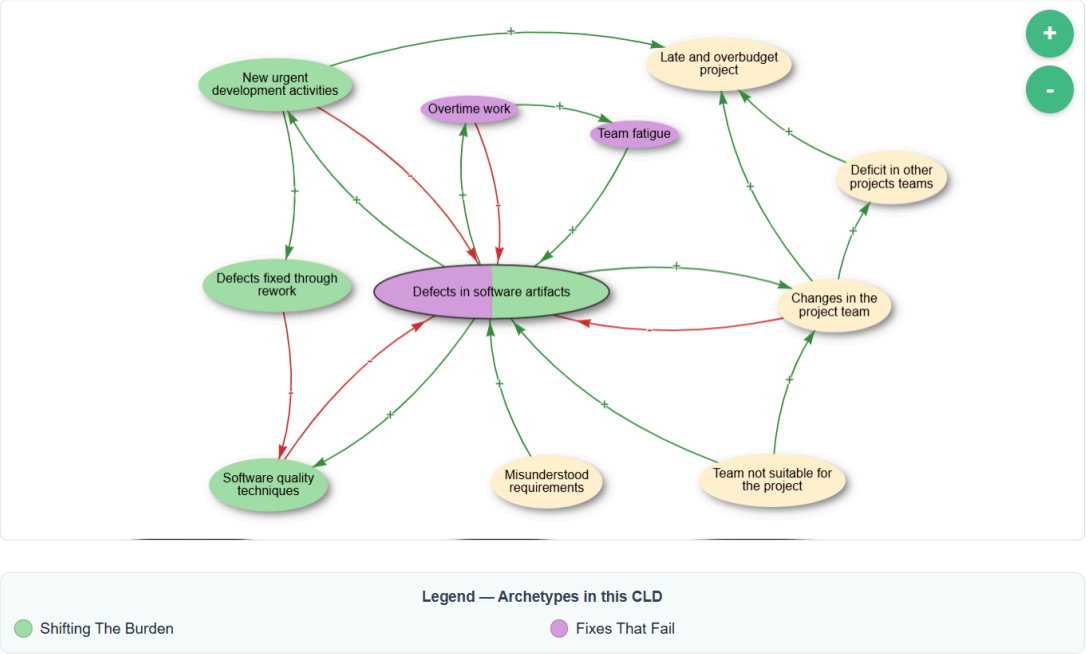

Figure 41. CLD with overlapping archetypes.

CaLMo has analytical capabilities that extend beyond visualization. In a CLD, by clicking on variables, loops, or archetypes, the user accesses details about them, which can help analyze the CLD and understand the behaviors it represents. For example, when the user clicks a variable that is part of at least one feedback loop or archetype, the tool shows the variable details view (Figure 42), illustrated for *Defects in software artifacts*, listing the reinforcing and balancing loops and the archetype instances that include the selected variable, together with the variables that compose each structure.This allows, for example, inspecting central variables with many relationships – such as Defects in software artifacts in the illustrative scenario – and seeing at a glance which feedback structures and archetypes they participate in and how they connect to the rest of the system.

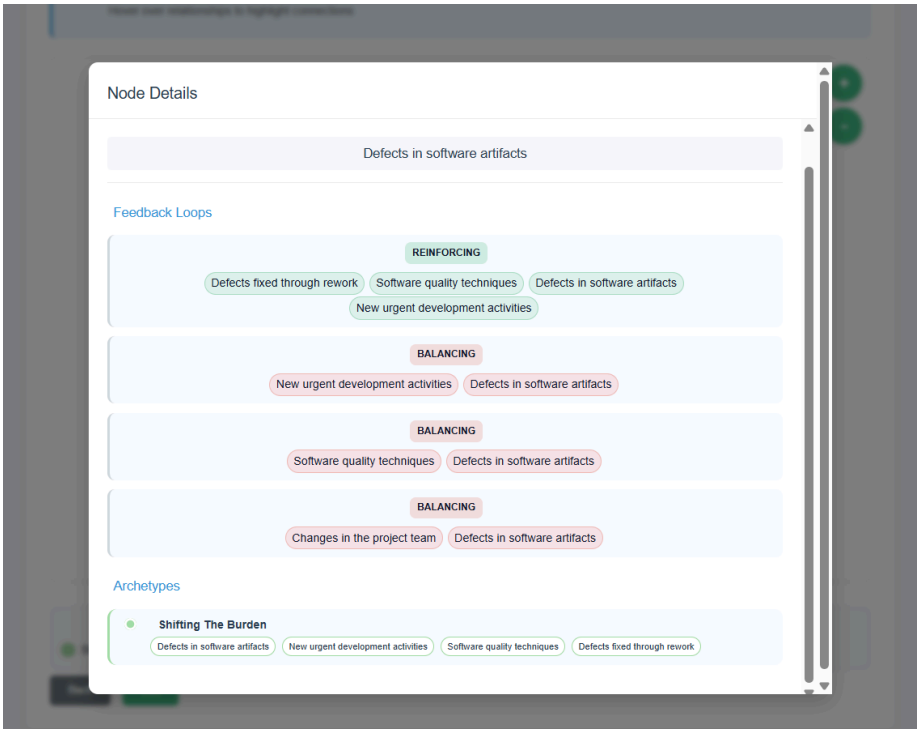

Figure 42. CLD elements detail.

All created variables, relationships, and CLDs persist in CaLMo's repository, which organizes CLDs for easier access and management (Figure 43). From this CLD library, users can view, edit, or delete a CLD, as well as create new ones. A user only has access to the CLDs and variables created by them. In this way, a user does not interfer in variables and CLDs created by others. By selecting Edit, the user is taken to the Edit CLD view (Figure 44), where they can update the CLD's basic information (name, description, and date) and iteratively refine its causal structure by adding, removing, or modifying relationships, including their source, target, and polarity. This functionality supports maintaining and evolving CLDs over time.

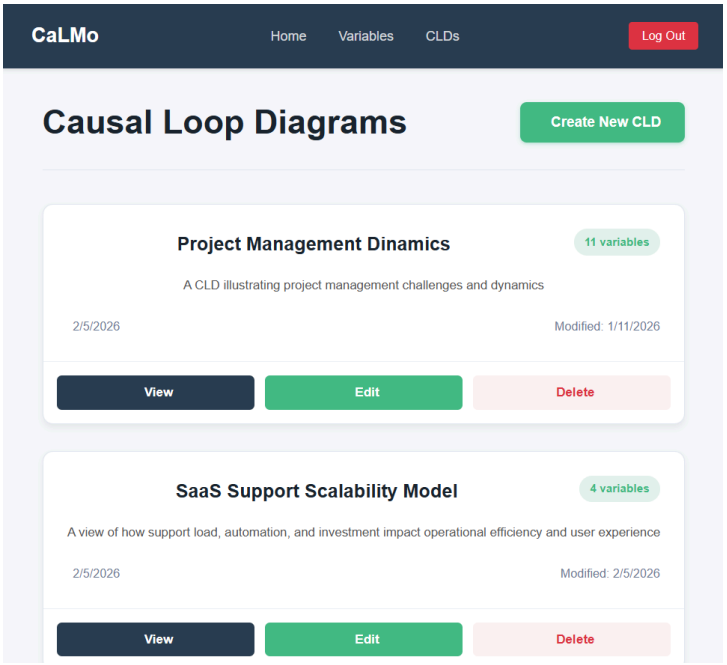

Figure 43. CLD library (stored CLDs and available actions).

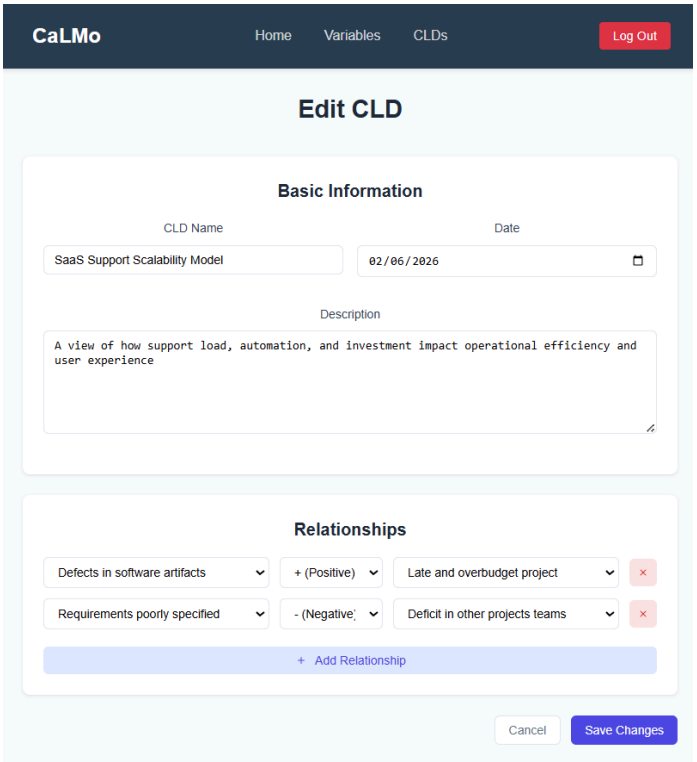

Figure 44. Editing an existing CLD.

Finally, CaLMo includes a *Getting Started* page that acts as an onboarding guide, combining conceptual explanations of CLD elements with step-by-step guidance. It briefly introduces CLDs, presents a simple example, and summarizes the main concepts used in the tool (variables, causal links and polarity, feedback loops, and archetypes). The page also outlines the recommended workflow for creating and analyzing CLDs in CaLMo, supporting the user during the modeling process, especially for those who are using the tool for the first time or are not very familiar with CLDs.

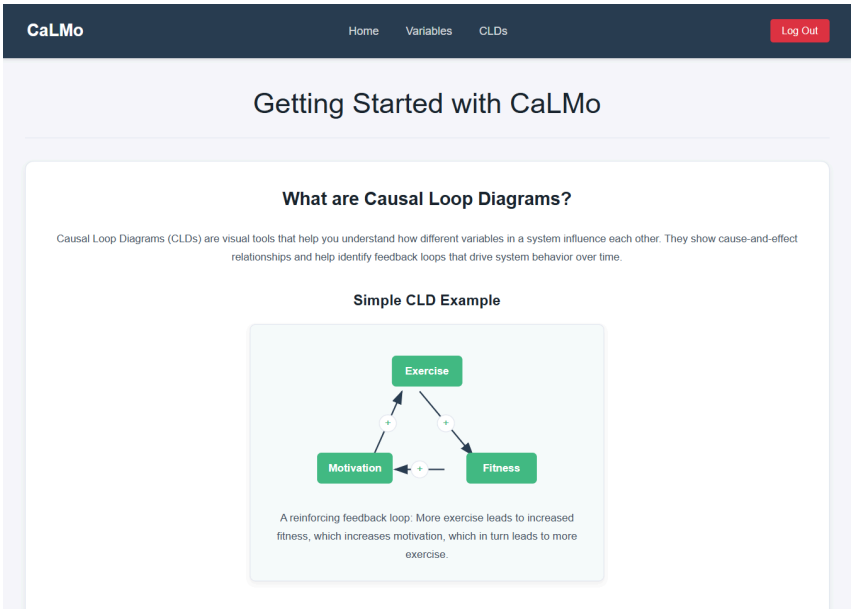

Figure 45. Getting Started page.

3.2 CaLMo Limitations and Future Improvements

In this section we point out limitations of CaLMo and some planned actions to address them and improve the tool.

The archetype-identification engine matches only canonical (“by-the-book”) patterns, searching for structures that replicate the basic structure for each archetype. More complex archetypes, containing mediating variables are currently not recognized. In addition, although delays are conceptually central in several archetypes, they are not explicitly represented or exploited by the current engine, which operates only on variables and relationships.

These restrictions point to concrete directions for future work. On the analytical side, the engine will be extended to recognize pattern variants and to incorporate explicit delay representations. Moreover, CaLMo could be extended to support user-defined archetypes, allowing users to specify new structural patterns to be detected by the tool, thus increasing its extensibility and adaptability to domain-specific contexts.

On the visualization side, CaLMo is being evolved to annotate feedback loops labels directly on the CLD canvas (e.g., R1, B1, B2), avoiding the need to open the variable-details view to locate them.

Concerning CLD creation, CaLMo currently adopts a form-based workflow. Users define variables (via CRUD forms) and specify relationships between them in dedicated pages, and then CaLMo automatically generates the CLD and shows it to the user in a new page. This interaction style requires users to switch pages to see the resulting diagram and verify whether archetype instances were identified. Moreover, the form-based workflow may be less convenient for users who prefer to construct CLDs by drawing directly on the canvas. We are working on an improvement to provide a visual editing mode in which users add variables and relationships using graphical constructs directly on the CLD canvas, but still preserving CaLMo's underlying structured representation. This will also enable a more immediate, incremental experience, with the CLD progressively rendered and archetypes dynamically detected as elements are added.

Beyond improving the interaction during modeling, we also plan to develop new features to support collaborative work and provide advances to enable interoperability with other tools (e.g., import a CLD produced in another tool, identify its elements, and store the CLD in the CaLMo library).

FINAL CONSIDERATIONS

Throughout this document, we discussed how System Thinking can contribute to a broader and deeper understanding of environments in Software Engineering. By adopting this perspective, professionals and researchers can identify not only individual causes of problems but also the systemic structures that sustain them over time.

We introduced the Causal Loop Diagram as one of the main tools supporting this approach. CLDs allow the representation of causal relationships, feedback loops, and archetypes that help explain dynamic behaviors within organizations. Through these models, it becomes possible to visualize how processes, practices, and people interact, revealing leverage points for meaningful and sustainable improvement.

Applying System Thinking is not only about modeling systems. It is about developing the ability to see connections, understand complexity, and design better interventions in a world that increasingly demands holistic perspectives. CLD is a powerful tool in this context. This document provided theoretical knowledge and practical guidance on the use of CLD in Software Engineering. We expect to support software engineers, researchers, and students to take a step further on exploring the use of CLD to address software engineering problems.

REFERENCES:

APOLINÁRIO, Amanda Brito; LAHASS, Thiago Felippe Neitzke; BORGES, Júlia de Souza; SANTOS JÚNIOR, Paulo Sérgio dos; BARCELLOS, Monalessa P.. CaLMo: A Tool to support the use of Causal Loop Diagram in Software Engineering. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE (SBES), 39., 2025, Recife/PE. Porto Alegre: Sociedade Brasileira de Computação, 2025 . p. 969-975. ISSN 2833-0633. DOI: https://doi.org/10.5753/sbes.2025.11543.

FATEMA, I.; SAKIB, K. Factors influencing productivity of agile software development teamwork: A qualitative system dynamics approach. In: IEEE. 24th Asia-Pacific Software Engineering Conference (APSEC). [S.l.], 2017. v. 2017-December, p. 737–742.

FATEMA, I.; SAKIB, K. M. U. Analyse agile software development teamwork productivity using qualitative system dynamics approach. ICSEA 2017, p. 71, 2017

JÚNIOR, P. S. dos S.; BARCELLOS, M. P.; CALHAU, R. F. First step climbing the stairway to heaven model-results from a case study in industry. Journal of Software Engineering Research and Development, v. 10, p. 5–1, 2022.

KIM, Daniel H.; ANDERSON, Virginia. Systems archetype basics. Waltham, Mass, Pegasus Communications Inc, 1998.

Kim, D. (2018). Introduction to System Thinking. Pegasus Communications.

LAHASS, Thiago Felippe Neitzke; BORGES, Júlia de Souza; APOLINÁRIO, Amanda Brito; SANTOS JÚNIOR, Paulo Sérgio dos; BARCELLOS, Monalessa P., 2026, A Causal Loop Diagram supporting tool to bridge System Thinking and Software Engineering, Journal of Software Engineering Research and Development (under review).

Meadows, D. H. (2008). Thinking in systems: A primer. Chelsea Green Publishing.

Sterman J.D., Business Dynamics: Systems Thinking and Modeling for a Complex World, Irwin-McGraw Hill, Tata McGraw Hill Education Private Limited, 2010.