Change Is the Only Constant: Dynamic Updates for Workflows

Daniel Sokolowski
daniel.sokolowski@unisg.ch
University of St. Gallen
Switzerland

Pascal Weisenburger
pascal.weisenburger@unisg.ch
University of St. Gallen
Switzerland

Guido Salvaneschi
guido.salvaneschi@unisg.ch
University of St. Gallen
Switzerland

ABSTRACT

Software systems must be updated regularly to address changing requirements and urgent issues like security-related bugs. Traditionally, updates are performed by shutting down the system to replace certain components. In modern software organizations, updates are increasingly frequent—up to multiple times per day—hence, shutting down the entire system is unacceptable. *Safe* dynamic software updating (DSU) enables component updates while the system is running by determining *when* the update can occur without causing errors. Safe DSU is crucial, especially for long-running or frequently executed asynchronous transactions (*workflows*), e.g., user-interactive sessions or order fulfillment processes. Unfortunately, previous research is limited to synchronous transaction models and does not address this case.

In this work, we propose a unified model for safe DSU in workflows. We discuss how state-of-the-art DSU solutions fit into this model and show that they incur significant overhead. To improve the performance, we introduce Essential Safety, a novel safe DSU approach that leverages the notion of *non-essential* changes, i.e., semantics preserving updates. In 106 realistic BPMN workflows, Essential Safety reduces the delay of workflow completions, on average, by 47.8 % compared to the state of the art. We show that the distinction of essential and non-essential changes plays a crucial role in this reduction and that, as suggested in the literature, non-essential changes are frequent: at least 60 % and often more than 90 % of systems' updates in eight monorepos we analyze.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; • **Applied computing** → Business process modeling.

KEYWORDS

Software Evolution, Dynamic Software Updating, Workflows

ACM Reference Format:
Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2022. Change Is the Only Constant: Dynamic Updates for Workflows. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510065

1 INTRODUCTION

Updating long-running software systems is essential to address changing requirements and mitigate security vulnerabilities in a timely manner. Updates become more frequent in modern software development following agile methods and DevOps principles [19], requiring automation of updates and low impact on the running system to prevent frequent interruptions.

Traditionally, software updates are performed by shutting down software systems and restarting them after replacing some components with new versions. While this approach is simple, it is disruptive and infeasible for larger systems, where a full restart may take a long time. As a result, researchers have investigated less disruptive *dynamic software updating (DSU)* [52], i.e., updates that occur while the system is running. Thereby, a component update in the middle of a transaction must not result in inconsistencies. For example, if a client requires a security token to access a server and the server is updated to a new token scheme in a naïve way, verifying a previously generated token could fail [8]. Safe DSU determines *when* an update can be performed without incurring semantic inconsistencies—the so-called *update condition.* It leverages various information, such as the system topology and the progress of transactions.

A common solution to implement long-running, frequent, and expensive transactions are *workflows*, sometimes referred to as *long-running transactions* [27, 40]. Workflows are extremely common in modern software applications to express a sequence of tasks and the data flow between them, decoupling process flows from application logic and enabling automation. Workflows have been used for a long time and recently gained popularity to orchestrate weakly coupled components. For instance, workflow engines have been adopted at modern software companies (e.g., Conductor at Netflix [42]) and are supported by major cloud providers (e.g., AWS StepFunction [4] and Google Cloud Workflows [26]).

Safe DSU is crucial for workflows: Ignoring update safety and retrying the transactions that were broken by an update may be acceptable for applications with inexpensive, short-lived transactions, but the cost of repeating broken transactions increases with the transactions' amount, duration, and resource consumption. Thus, especially for long-running or frequently executed workflows, delays or retries after failure on component updates potentially require large amounts of additional resources and introduce severe delays. This issue is even more relevant in CI/CD pipelines, where changes are small and frequently deployed [15].

Unfortunately, existing safe DSU solutions have not been studied in the context of real-world workflows. Crucially, previous research only considers synchronous transactions and cannot be directly transferred to workflows, which are asynchronous. To close this gap, we investigate a new formal model for safe DSU suitable for workflows. We show how our new model can capture state-of-the-art

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510065

DSU approaches and analytically compare them within the model, setting the conceptual ground for the performance differences that we later inspect empirically.

Another challenge in using safe DSU for workflows is that existing approaches introduce significant performance overhead. They either do not reach their update condition in a timely manner (Version Consistency (VC) [8, 38]) or make strong assumptions sacrificing safety if the assumptions are not satisfied (Tranquility (TQ) [60]). To reduce the performance overhead compared to the state of the art *and* retain safety, we propose the safe DSU approach Essential Safety. Its update condition Essential Freeness is based on the observation that a significant amount of updates are *non-essential* changes, i.e., they never interfere with running transactions because they do not introduce semantic changes. Thanks to identifying non-essential changes, Essential Safety reduces delays and interruptions due to updates in workflow-based applications and retains strong guarantees on correct system operation.

This work paves the way to apply safe DSU to modern workflow architectures, e.g., in cloud applications [4, 26]. In addition, the identification of non-essential changes—updates without semantic changes—can be refined using insights from the application and the developers. This insight opens an opportunity for future research to provide more precise characterizations of non-essential changes, reducing the number of expensive updates and further improving DSU performance.

All our evaluation data and the software developed for this paper are publicly available [56]. In summary, this paper makes the following contributions:

(1) We propose a new formal model for DSU supporting asynchronous workflows. We show that state-of-the-art DSU approaches, as well as our approach, fit into such a model, enabling a direct comparison.
(2) We propose Essential Safety as a novel approach for safe DSU, which leverages the identification of whether an update introduces a semantic change, i.e., is essential.
(3) We analytically compare Essential Safety to previous DSU approaches, show that Version Consistency is a conservative over-approximation of Tranquility and Essential Safety, and highlight the different information taken into account, explaining the performance difference among such solutions.
(4) We empirically confirm by simulating 106 realistic collaborative BPMN workflows and analyzing eight monorepos that Essential Safety provides the best performance among safe DSU approaches, that identifying non-essential changes is effective to improve safe DSU's performance, and that, in practice, at least 60 % and often more than 90 % of the updates are non-essential changes.

The paper is organized as follows. Section 2 outlines the issue of safe DSU for workflows. Section 3 presents related work. Section 4 describes our model for safe DSU in workflows and introduces our approach Essential Safety. Section 5 presents Essential Safety's practical realization. Section 6 analytically compares Essential Safety with previous DSU solutions. Section 7 empirically evaluates our contribution, and Section 8 concludes.

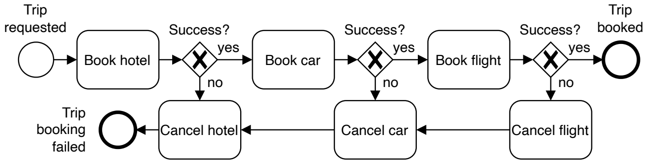


Figure 1: BPMN workflow of the trip booking saga.

2 THE DYNAMIC UPDATE PROBLEM

In this section, we introduce a workflow as a running example to explain the problem of safe DSU. Workflows are used extensively in software systems to express the execution of interrelated tasks.

2.1 The Trip Booking Saga

The running example is the trip booking case study [48], shown as BPMN workflow [45] in Figure 1. A hotel, a car, and a flight are booked sequentially. Each of these steps may fail, triggering compensating actions for the bookings performed up to the current execution point—a design pattern referred to as "saga" [22].

Each task in the workflow is implemented as a serverless function. Some of the functions are coupled through a shared database on which they operate, constituting components. In our example, the car booking and cancel functions constitute the *car rental* component, and the remaining four functions are the *holiday* component. These components are the smallest unit of updates, e.g., when the car rental component is updated, the serverless functions for both "book car" and "cancel car" are replaced by a new version.

Figure 2 shows the trip booking case study as a UML sequence diagram. We added the labels A to E and b to d to reference points in time during the execution. If there is no error, only A to E occur and not b to d, because they are on the paths that only occur on the failure of a booking task, executing the compensation tasks.

2.2 The Need for Safe Dynamic Updates

Updating a component in a workflow may break the correct execution in two cases. The first case is the update of a component while it is currently executing a task, i.e., it is active. For example, if a workflow instance runs the "book hotel" task, updating the holiday component can cause incorrect behavior. In line with the literature on dynamic updates, we consider updating an active component (i.e., one that executes a task) always unsafe—this problem is studied in a different research line [18, 58, 59] and requires *hot-swapping* code as well as migrating the state representation across versions.

The second case is when a component performs two tasks within the same transaction and an update introduces a semantic change in between. For example, in the trip booking saga, after "book hotel" completes (after B in Figure 2), if "book car" is not successful and the holiday component is updated with a new version that uses a different format for hotel booking IDs, "cancel hotel" does not behave correctly: either it does not find the correct booking to cancel or—even worse—it finds the wrong one. Thus, the workflow instance fails to revoke the hotel booking. To avoid such errors, safe DSU approaches specify *update conditions*. They determine *when* an update can be performed such that it does not cause semantic mismatches.

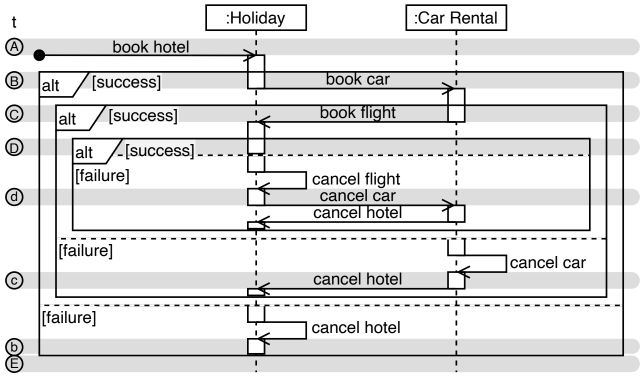


Figure 2: Sequence diagram of the trip booking saga.

2.3 The Role of Non-Essential Changes

To reach the update condition for a safe update and to uphold it until the update completes, safe DSU approaches require monitoring and may block tasks. For instance, to update the car rental component, a DSU approach might block all calls to the car rental component until the update completes. Once all running tasks on car rental terminate, a safe update condition is met and upheld until the update is completed. Both reaching and upholding a safe update condition lead to significant overhead and delay. The overhead grows with the amount, duration, and resource consumption of the transactions, which results in considerable overhead for highly frequent, expensive transactions that can be found in workflows. DSU approaches should block tasks as little and short as possible.

For existing DSU approaches [32, 38, 60], the overhead is high in long-running, frequently executing workflows (cf. Section 7), prohibiting their use for the safe continuous deployment of such applications. However, in practice, a substantial fraction of the changes running through a continuous deployment pipeline tend to be small and do not introduce semantic changes, i.e., *non-essential changes*—a reality ignored by previous work on DSU. Thus, we can apply a less disruptive update condition to most updates. It requires less task blocking to be reached and upheld and greatly reduces unnecessary overhead.

Essential Safety (ES), our novel safe DSU approach for workflows, leverages the distinction between essential and non-essential changes. Essential Safety reduces DSU disruption to a minimum while providing the same safety as the state of the art, enabling safe DSU in real-world, long-running, frequently executed workflows.

In Table 1, we compare, for existing DSU approaches [32, 38, 60] and for Essential Safety, *when* updating the components in the trip booking case study (cf. Section 2.1) is safe. Checkmarks correspond to safe update time intervals. The highlighted cells indicate intervals where the component is active—these intervals are unsafe under all update conditions. If an update is an essential change, Essential Safety provides the same safe update intervals as Version Consistency (VC), which is generally safe—in contrast to Tranquility (TQ). For non-essential changes, the intervals indicated by (✓) are additional safe intervals and, thus, Essential Safety provides the highest number of safe update intervals.

Table 1: Update conditions over the time *t* in the trip booking saga for Quiescence (Q), Tranquility (TQ), Version Consistency (VC), and Essential Safety (ES).

	Holiday				Car Rental			
t	Q	TQ	VC	ES	Q	TQ	VC	ES
A		✓	✓	✓		✓	✓	✓
B	▨	▨	▨	▨		✓	✓	✓
C		$\cancel{\checkmark}$		(✓)	▨	▨	▨	▨
D	▨	▨	▨	▨				(✓)
d	▨	▨	▨	▨				(✓)
c		$\cancel{\checkmark}$		(✓)	▨	▨	▨	▨
b	▨	▨	▨	▨	✓	✓	✓	✓
E	✓	✓	✓	✓	✓	✓	✓	✓

3 RELATED WORK

Closest to our work is previous research on opaque box safe DSU [8, 32, 38, 60], which we compare with in detail in Section 6. In contrast, transparent box approaches [29, 57] leverage formal models of the programs to identify points in time when it is safe to update. While transparent box methods allow more fine-grained analyses, they rely on strong assumptions on the implementation technology, making them hard to apply to general distributed systems, supporting components implemented using heterogeneous paradigms, languages, and technologies. The survey of Seifzadeh et al. [52] discusses several platforms with DSU. We now discuss related research on (1) updating software in running processes, (2) software reconfiguration, (3) workflow evolution, and (4) continuous delivery. Finally, we provide an insight into (5) DSU in practice.

Dynamic Code Replacement. In this work, we focus on safe dynamic updates of components that are *not* currently executing code. Complementary to our work, there are approaches for updating running components. Updating the code of a running program was investigated already in the 1970s [18]. Later, Erlang [6] has been one of the first programming languages to enable *hot swapping*, i.e., modules can be replaced at run time (the new version is loaded when the next invocation occurs), and programmers can specify state transfer between modules. A similar solution for dynamic code replacement is also available in the Ada programming language [59]. More recently, dynamic code replacement on the Java Virtual Machine has been supported in the Jvolve [58] and in the DCE VMs [61] as a modification to the Java HotSpot VM.

These approaches focus on the technical realization of code replacement while the system is running and assume that developers correctly handle transferring the state of components across updates. Another line of work focuses on ensuring that state transformations are correct, e.g., using type systems [30]. Gu et al. [28] replay the sequence of invocations performed on the old object on the new one to ensure that it reaches the same state.

Software Reconfiguration. Dynamic software reconfiguration is about changing the configuration of a software product at run time while the system is operational. Research focuses on reconfiguration models ensuring the preservation of consistency properties and minimizing system disruption [11]. Adapta [49] is a reflective middleware for self-adaptive, component-based applications. It aims to decouple the application logic from the code that handles the adaptation, and it requires run-time monitoring and triggering mechanisms. Software reconfiguration has been applied to distributed execution, where remote system nodes interpret reconfiguration scripts [10]. Software self-adaptivity is a research line on switching the behavior of applications at run time, for example, using metaprogramming or reconfiguration of component-based systems [39].

Updating Workflows. Researchers have examined how existing workflows can be modeled to support changes while they are executed. Casati et al. [13] address the problem by defining a set of transformation rules and dividing the state space into parts that are terminated or handled by different process definitions. Geiger et al. [23, 24] present a detailed review of the current state and evolution of BPMN 2.0 support and implementation, finding a lack of standard compliance in current implementations.

Updates in Continuous Delivery. A recent overview of the impact of continuous delivery (CD) by Lwakatare et al. [37] investigates CD implementations in five different development contexts. Laukkanen et al. [34] provide an overview of adoption problems of CD and show that most research work focuses on issues that developers face, but developers usually consider release and software update problems to be external factors. Updating components in CD poses a problem in real-world settings according to semi-structured interviews conducted by Claps et al. [14] at Atlassian. At least 7 out of 10 interviews highlight that *seamless upgrades* are hard to implement in large systems and potentially consume significant amounts of resources. Gallaba et al. [21] infer dependencies between components using build execution tracing to accelerate CI/CD pipelines—information that could be used to identify non-essential changes. Infrastructure as code (IaC) has been adopted to increase automation in modern development pipelines [41]. Traditionally, IaC solutions in CD are executed as one-off tasks and treat *when* to update as external decision. However, the recent introduction of long-running, reactive deployment scripts [55] blurs the line between application and infrastructure code, enabling the required monitoring and logic for safe DSU.

DSU in Practice. Today, safe DSU relies on complex workarounds, avoiding the need for safe update intervals. Cloud vendors and deployment platforms, e.g., Kubernetes [1], provide variations of blue-green [20] and canary [50] deployment strategies. Parallel change [51] is a pattern for safe interface updates that replaces unsafe changes with a sequence of safe ones. These solutions provide safe DSU for software where the components for the application logic are stateless and a (transactional) database holds the state. However, this hypothesis does not always apply, e.g., in the case of workflows involving components that belong to various authorities. In such a case, using a central database is infeasible—a codified principle in microservice architectures [43]. In many scenarios, e.g., Web applications for social networks, it is accepted that updates may break multi-request transactions—retry is cheap, but it hampers user experience. In other scenarios, retry is not acceptable because it requires too much time or resources, wherefore safe DSU is needed to minimize the updates' impact.

4 EFFICIENT, SAFE DYNAMIC UPDATES OF WORKFLOW COMPONENTS

In this section, we present a formal model for workflow execution. We propose Essential Safety as a safe and efficient updating approach. We then show how Essential Safety's update condition Essential Freeness can be reached and upheld during an update.

4.1 Workflow Execution Model

Various workflow modeling languages exist, including standards like BPMN [45] and BPEL [44], as well as vendor-specific DSLs like the Amazon States Language [3] for AWS Step Functions [4]. Though these modeling languages differ in features and expressivity, they all organize consecutively executed tasks in a graph structure. We formally model their shared core concepts.

We consider the system landscape $L = (\mathcal{R}, \mathcal{W}, \mathcal{T}, C, i)$ consisting of workflow engines $\mathcal{R}$, workflows $\mathcal{W}$, and tasks $\mathcal{T}$, which are implemented by components C, related by $i : \mathcal{T} \to C$. We model a workflow $W = (T, P, B, E) \in \mathcal{W}$ with a directed graph of tasks $T \subseteq \mathcal{T}$ that are connected to the tasks which can be executed next—the *succeeding* tasks—by arcs $P \subseteq T \times T$. A workflow's initial tasks are $B \subseteq T$ and the end tasks $E \subseteq T$. All non-end tasks must have at least one succeeding task. Thus, a task $t \in T$ is either in E or there exists at least one edge $(t, t') \in P$. Workflows are executed as workflow instances $I = (r, W, A, V, F, S) \in \mathcal{I}$ in the workflow engine $r \in \mathcal{R}$ where $A \subseteq T$ is the active tasks, initialized as $A = B$. The workflow engine updates A during the execution of I. The workflow instance terminates once no task is active anymore, i.e., $A = \emptyset$. S is the workflow instance's state. All tasks T of I can read from it at the beginning of their execution and write to it after their execution. $V \subseteq T$ are the visited tasks, i.e., the set is initially empty, and all tasks that are removed from A during the execution are added to V. $F \subseteq T$ are the potential future tasks, i.e., all tasks that are reachable in the directed graph (T, P) from a task in A. Note that F is a conservative over-approximation of the future tasks, i.e., not all tasks in F have to be executed. For instance, consider BPMN exclusive gateways, as included three times in Figure 1, which have multiple outgoing paths, but only exactly one will be executed. All tasks on a path after a BPMN exclusive gateway are initially in F and all tasks on the paths not taken are removed from F without being executed once the gateway is processed.

Based on the definition of active tasks A, we define a component $c \in C$ is active if it executes any active task:

Definition 4.1 (Active Component). A component $C \in C$ is called *active* if it currently executes a task in any workflow instance: $\exists I = (r, W, A, V, F, S) \in \mathcal{I}, t \in A : i(t) = C$.

4.2 Essential Safety

We define a safe and efficient update condition for long-running, frequently executed workflows. A workflow instance always executes correctly if every component is only updated (1) after it has executed its last task, (2) before it executes its first task, or (3) if it does not execute any task in the workflow instance. In contrast to previous work, our approach also allows a component to be updated if it already executed a task *and* may execute a task in the future if the update does not introduce a semantic change. We call such updates *non-essential changes*, in contrast to *essential changes*, which introduce a semantic modification:

Definition 4.2 (Essential Change). An update of a component $C \in \mathcal{C}$ from version v to v' is an essential change for workflow instance $I = (r, W, A, V, F, S) \in \mathcal{I}$, if the possible execution of any future task $t \in F \mid i(t) = C$ on v' is not guaranteed to produce the same resulting state S and side effects as executing t on v.

Every other change (given the definition above) is a non-essential change. Identifying whether a change is non-essential is not decidable in general as it boils down to the program equivalence problem. Since misclassifying essential changes as non-essential breaks safety, we conservatively under-approximate non-essential changes with a catalog of known non-essential changes that can be found through efficient analyses. Kawrykow and Robillard [31] describe non-essential changes as (1) cosmetic, (2) behavior-preserving, and (3) unlikely to provide further insight into component relationships. This includes—but is not limited to—trivial type updates, local variable extractions, rename-induced modifications, trivial keyword modifications, local variable renames, and whitespace and documentation-related updates. Definition 4.2 leaves open adding more sophisticated analyses to find non-essential changes, including application-specific ones. Identifying non-essential changes is important in practice but orthogonal to our contribution.

Updating a component with non-essential changes is always safe while the component is not active. We introduce Essential Safety (ES): only updating components when they are *essentially free*.

Definition 4.3 (Essential Freeness). A component $C \in \mathcal{C}$ is *essentially free*, if it

(1) is not active and
(2) a. will not be active in a workflow instance in which it already executed a task ($\nexists I = (W, A, V, F, S) \in \mathcal{I}, t \in V, t' \in F : i(t) = C \wedge i(t') = C$) or
  b. its update is a non-essential change for all workflow instances $I = (W, A, V, F, S) \in \mathcal{I}, t \in V \mid i(t) = C$ in which it already processed a task.

Considering a single workflow instance of the trip booking saga, updates with non-essential changes can always be performed without violating the workflow's correctness if the respective component is not currently executing a task. For instance, using the intervals marked in Figure 2, the car rental component can always be updated except within C and c. If the update is an essential change, the update must not occur between a component's first and last task execution in the workflow instance. For example, an essential change of the car rental component may not occur within B, d, and c because it might be the case that "cancel car" is executed in the future after that "book car" has been already executed on the current version of the component.

4.3 Reaching Essential Freeness

Strategies to reach safe DSU update conditions trade-off between update *timeliness* and *interruption*. Timeliness is the length of the interval between requesting the update and the beginning of the component exchange, i.e., the point in time when the component stops executing tasks. Interruption is how long a workflow instance's completion is delayed due to the update. The following reaching strategies from the literature [32, 38, 60] can be used to reach Essential Freeness.

Waiting (W). The update waits for Essential Freeness. The interruption is limited because only workflow instances that started after the update begins are delayed and the update's duration bounds the interruption. Yet, the update is not guaranteed to start in bounded time, i.e., timeliness is unpredictable. Thus, this approach is not suitable where Essential Freeness rarely occurs by chance.

Blocking Tasks (BT). The starting of tasks on the component to update is delayed until after the update. This strategy ensures that Essential Freeness is reached in bounded time, but it may cause more interruption than Waiting.

Blocking Instances (BI). The strategy is similar to Blocking Tasks, but instead of delaying tasks, the start of new workflow instances that need the component is delayed until after the update. While this strategy also guarantees the update is reached, it might take longer. The interruption is expected to be similar to Blocking Tasks but is reduced if multiple updates occur in parallel.

Concurrent Versions (CV). For non-essential changes, all new task executions are served by the new version running in parallel to the old version, which completes the already running tasks. For essential changes, the old version also executes new tasks belonging to workflow instances that already executed at least one task on it. Thus, the old version remains available until no workflow instance needs it anymore. This strategy provides good timeliness and no interruption, but it requires running two parallel versions of a component, significantly increasing complexity, especially for stateful components.

Except for Concurrent Versions, all reaching strategies require Essential Freeness to hold until the update is completed. This can be achieved by applying Blocking Tasks to a component during its update, delaying the start of new tasks until the update is completed.

5 REALIZING SAFE DYNAMIC UPDATES

Determining and reaching the update condition is trivial with a single centralized workflow engine. Such a central entity (1) knows the state of all workflow instances $\mathcal{I}$, and (2) can delay the execution of tasks and whole workflow instances. However, the system landscape $\mathcal{L}$ may comprise multiple workflow engines $\mathcal{R}$, each hosting a subset of workflow instances $\mathcal{I}$. Hence, no centralized view exists on all workflow instances. Modern, scalable workflow engines, e.g., Zeebe [12], are by default decentralized over multiple separate workflow engines to improve scalability and fault tolerance.

To ensure their safe update, all workflow engines invoking tasks on a *shared* component have to coordinate. Hence, reaching Essential Freeness for a component requires considering all workflow instances $\mathcal{I}$ that use the component. We propose a dissemination algorithm that the workflow engines use to notify components of their workflow instances' status. The algorithm ensures that components are aware of their role in all workflow instances using them. Each component can then locally decide whether it reached the update condition and can be safely updated.

5.1 Dissemination Algorithm

Algorithm 1 shows the dissemination algorithm that workflow engines execute *for each workflow instance*. The four callback procedures in Algorithm 1 are called reactively based on the events in the workflow execution, e.g., before a task is started, the procedure in Line 1.5 is called. Using the procedures, the workflow engine (1) *announces* to components that they might be used, (2) *marks* components that were used, and (3) *locks* components (not exclusively) while they are used. Every component stores its status in the workflow instances, i.e., every component maintains for each workflow instance the information whether it received an announcement or a marking, which was not revoked yet, and a lock counter.

BeforeWorkflow, AfterWorkflow, BeforeEachTask, and AfterEachTask are executed on the workflow engine before/after a workflow instance is executed and before/after each task is run. The Announce, RevokeAnnouncement, Mark, RevokeMarking, Lock, and Unlock procedures are called remotely on the component passed as their first argument. Remote calls are asynchronous unless the execution blocks to get the return value using `await`. Components can delay their response at the `await` synchronization points to interrupt the workflow instance's execution until they can accept the announcement, marking, or lock.

The BeforeWorkflow procedure (Line 1.2) announces to components c (Line 1.4) from the set of potential future tasks (Line 1.3) of the workflow instance I that c might participate in I. Announcements are revoked after completing tasks (Lines 1.11 to 1.14) if the component will not be used (again).

Before the workflow instance invokes a task on a component for the first time, the engine marks the component (Lines 1.6 to 1.8). Markings remain for the rest of the workflow instance's execution and are revoked after its completion (Line 1.17).

Every time a workflow instance invokes a task, the engine locks the component (Line 1.9) and unlocks it after the task is completed (Line 1.15). A workflow instance might lock the same component multiple times before unlocking due to parallel task execution. The components internally increase a counter with every locking and decrease it with every unlocking. If the counter is positive, the workflow instance runs tasks on the component.

The presentation of Algorithm 1 is simplified for clarity. Our implementation includes some optimizations, e.g., it sends announcements (Line 1.4) in parallel, and the messages to $i(Task)$ in Lines 1.5 to 1.9 are packed into a single multipurpose message.

Algorithm 1 Modular dissemination algorithm on workflow engine r for workflow instance (r, W, A, V, F, S) with unique identifier I.

```
            Announcements        Markings        Locks
 1: Announcements, Markings ← $\emptyset$
 2: procedure BeforeWorkflow
 3:     Announcements ← $\{c \in C \mid \exists t \in F : i(t) = c\}$
 4:     for all $c \in$ Announcements do await Announce($c, I$)
 5: procedure BeforeEachTask(Task)
 6:     if $i$(Task) $\notin$ Markings then
 7:         await Mark($i$(Task), $I$)
 8:         Markings ← Markings $\cup$ $i$(Task)
 9:     await Lock($i$(Task), $I$)
10: procedure AfterEachTask(Task)
11:     PreviousAnnouncements ← Announcements
12:     Announcements ← $\{c \in C \mid \exists t \in F : i(t) = c\}$
13:     for all $c \in$ PreviousAnnouncements \ Announcements do
14:         RevokeAnnouncement($c, I$)
15:     Unlock($i$(Task), $I$)
16: procedure AfterWorkflow
17:     for all $c \in$ Markings do RevokeMarking($c, I$)
```

5.2 Handling Essential Freeness

Algorithm 1 disseminates the necessary information to the components to determine, reach, and uphold Essential Freeness. When performing an update that introduces essential changes, a component is essentially free if it holds for no workflow instance an announcement *and* a marking, i.e., no workflow instance that already used the component will use it again. For non-essential changes, a component is essentially free if it is not locked, i.e., for no workflow instance, the lock counter is greater than zero.

Using the Blocking Instances strategy, a component delays the confirmation of announcements until the update has been completed, blocking all new workflow instances that will call the component in BeforeWorkflow and delaying their start. Blocking Tasks uses a similar approach with locks for non-essential changes and markings for essential changes. Locks block all workflow instances, markings only the ones that did not use the component yet. Blocking Tasks does not generally block all task invocations by delaying locks because this could lead to a deadlock in case of essential changes: The not-blocked workflow instances already used the component and may prevent Essential Freeness—which would cause a deadlock—or do not use it anymore. For this reason, Blocking Tasks can only be activated at one component at a time to ensure deadlock-freeness.

For both strategies above, once reached, the update condition must be upheld until the update completes. For essential changes, delaying the confirmation of markings upholds the update condition. For non-essential changes, locks are delayed.

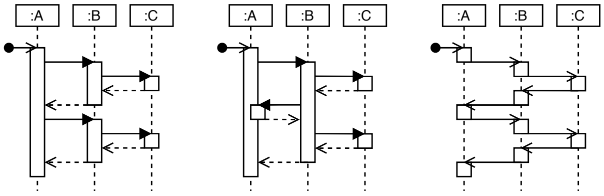
Waiting and Concurrent Versions do not require any aspect of Algorithm 1 for their update condition because they do not influence the execution of workflow instances.

6 WORKFLOW UPDATES IN CONTEXT

In this section, we show how Essential Safety relates to existing solutions for safe dynamic software updating.

6.1 From Transactions to Workflows

Previous work on DSU [8, 32, 38, 60] focused on synchronous distributed transactions in component-based systems. They assume that external clients trigger so-called root transactions, which, in turn, can run other (sub-)transactions on the same or other components. The execution blocks until a sub-transaction completes with a return value. In Figure 3a, component A runs a sub-transaction on B, which runs a sub-transaction on C. Afterward, C returns a value to B, B one to A, and new sub-transactions are run on B and C. Figure 3b shows the same interaction pattern but synchronous transactions are nested differently: Instead of ending the first transaction on B, B runs a sub-transaction on A.



(a) Synchronous transaction.

(b) Alternative synchronous transaction.

(c) Asynchronous workflow.

Figure 3: Modeling a synchronous transaction as workflow.

In this work, we model the system as asynchronous workflows. Tasks are started based on their order as defined in the workflow after the previous task(s) are completed. The tasks are coordinated by the workflow engine, which acts as an event-based middleware. Thus, upon completion, tasks send their results as an update of the state S to the workflow engine, which then starts the succeeding task(s) with S. Modern cloud-based systems—also beyond workflows—adopt the asynchronous model. Typically, such systems use asynchronous, decoupled communication patterns, e.g., event-based microservice choreographies or serverless computing [7, 35].

Our asynchronous workflow model can emulate the synchronous model. For instance, the transaction in Figure 3a can be modeled as in Figure 3c: the synchronous parent transactions are split into two tasks (before and after the subtransaction). State can be conveyed via the workflow instance's state S. This transformation is neither injective nor surjective: Not every asynchronous workflow can be translated to a synchronous transaction, and multiple differently nested synchronous transactions might be transformed to the same workflow, e.g., both Figures 3a and 3b result in Figure 3c.

6.2 Existing Safe DSU Approaches

We now present three update conditions from the literature and show how they fit into our asynchronous workflow model.

Kramer and Magee [32] proposed Quiescence (Q), which does not rely on run time information of workflow instances. Instead, the workflows' structure, which is known at design time, and whether a new workflow instance will be started suffice. If necessary, all future workflow instances are blocked to enforce Quiescence.

Definition 6.1 (Quiescence). A component is *quiescent* if it (1) is not active and (2) will not be active in a workflow instance.

Ma et al. [38] proposed Version Consistency (VC) and evaluated it with a simulation which was later extended with a system implementation and evaluation based on Apache Tuscany [8]. The update condition of Version Consistency is called Freeness.

Definition 6.2 (Freeness). A component is *free* if it (1) is not active and (2) will not be active in a workflow instance in which it already executed a task.

Version Consistency is similar to our approach but it does not distinguish between different types of updates, which are all conservatively over-approximated to an essential change.

Vandewoude et al. propose Tranquility (TQ) [60].

Definition 6.3 (Tranquility). A component is *tranquil* if it (1) is not active and (2) will not be active in a workflow instance in which it might execute a succeeding task for a component for which it already executed a succeeding task.

Though proposed before Version Consistency, Tranquility effectively corresponds to Version Consistency with the additional assumption that components follow a "black-box principle" [60]. For systems satisfying this principle, Tranquility assumes that version consistency does *not* have to be enforced between the internals of different sub-transactions. For instance, if within the same root transaction a client uses an authentication component and calls the server, which uses authentication internally, too, client and server may safely use different versions of the authentication component.

Leveraging the black-box principle, Tranquility results in better update timeliness and less interruption than Version Consistency [38]. However, Tranquility is unsafe for systems that do not follow the black-box principle, as Ma et al. [38] already noticed. It is generally questionable whether workflows follow the black-box principle because their tasks often depend on each other, leading to a violation of the principle.

Quiescence and Version Consistency map directly to our asynchronous workflow model (Section 4.1), i.e., Definitions 6.1 and 6.2 can be trivially verified by inspecting the potential future tasks F and the visited tasks V. Tranquility (Definition 6.3), however, distinguishes between sub-transactions that are called from the same transaction and ones that are called from a different transaction. For example, component C is tranquil between its two executions in Figure 3a, and it is not tranquil between its two task executions in Figure 3b. Yet, there is no such distinction in the asynchronous model in Figure 3c compared to the synchronous model (Section 6.1). Embracing Tranquility's black-box principle, we assume that all task executions of a component with preceding tasks from the same component belong to a single, synchronous transaction, i.e., for each component, the same version of another component is used for each of its succeeding tasks. With this definition, in the asynchronous workflow in Figure 3c, component C is only tranquil before the first and after the second task. Component A, however, may be updated after its first task, i.e., Tranquility is unsafe if components do not respect the black-box principle.

6.3 Update Conditions, Operationally

Different subsets of our dissemination algorithm (Algorithm 1) provide the necessary data to a component to (1) determine that an update condition for the component is reached, (2) reach the condition (quicker), and (3) uphold the condition once reached. Our algorithm is inspired by the control algorithm proposed for Version Consistency, which is based on graph transformations [38] and verified to be correct [8]. *Announcements* are similar to their *future edges* and *markings* to *past edges*; there is no counterpart for *locks*. However, their transaction model lacks a holistic view of transactions within the same root transaction because components do not share their internal logic. In our workflow model, such a view on workflow instances is available in their workflow engine. We leverage this holistic view to reduce communication. Table 2 summarizes the parts of Algorithm 1 required by the update conditions and reaching strategies. We now provide more detailed insights.

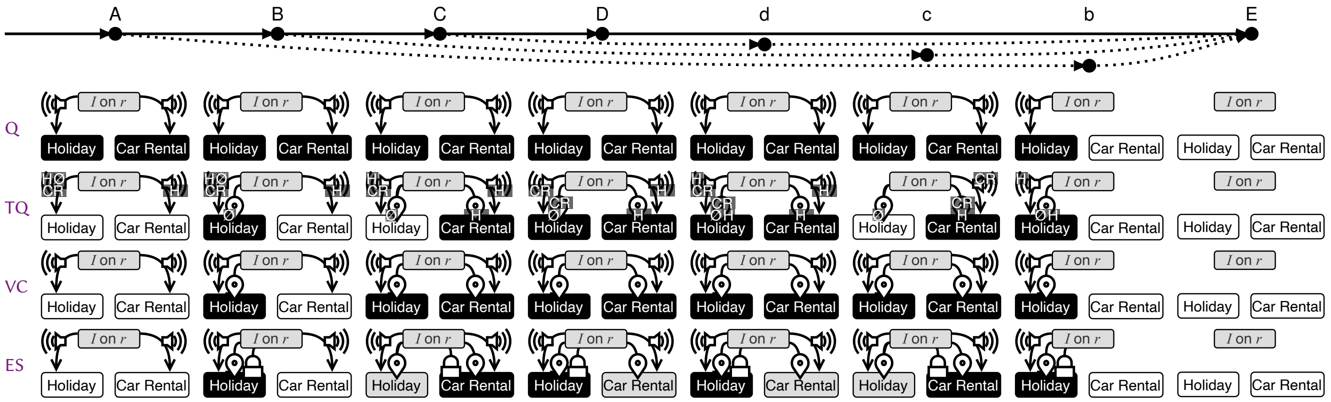


Figure 4: Announcements, markings, and locks at intervals in Figure 2 during the execution of the trip booking saga with Algorithm 1 for all update conditions (🔊 announcement, 📍 marking, 🔒 lock, update safe, non-essential update safe, update unsafe).

Table 2: Elements of Algorithm 1 for updating approaches and reaching strategies: ✓ necessary, ✓^{C} per preceding component, ✓* sufficient for non-ess. changes, ▲ uphold condition (ess. changes), ■ uphold condition (non-ess. changes).

Approach / Strategy	Announcements	Markings	Locks
Quiescence	✓ ▲ ■		
Tranquility	✓^{C}	✓^{C} ▲ ■	
Version Consistency	✓	✓ ▲ ■	
Essential Safety	✓	✓ ▲	✓ ■
Waiting			
Blocking Instances	✓		
Blocking Tasks		✓	✓*
Concurrent Versions		✓	

In Quiescence, announcements are sufficient to check if components are quiescent: They are if they have no announcements. To reach Quiescence, Kramer and Magee [32] only discuss Blocking Instances as "passivation" of components, i.e., ensuring that no transactions are invoked on the component in the future. The other three reaching strategies are also applicable but require run time information, which Kramer and Magee do not consider.

Version Consistency requires announcements and markings, but no locks. A component is free if it has for no workflow instance an announcement *and* a marking. Ma et al. [38] discuss Waiting, Blocking Tasks, and Concurrent Versions for Version Consistency, concluding that the last one should be the preferred strategy if applicable, otherwise Blocking Tasks. Essential Safety and Version Consistency are equivalent if all updates are essential changes.

Tranquility also requires announcements and markings. Yet, both need to be per workflow instance *and* per component of the preceding task(s)—not only per workflow instance as in Algorithm 1. The condition is then similar to Version Consistency: The component is tranquil if it has for no pair of workflow instance and preceding tasks' component an announcement *and* a marking. For Blocking Tasks, this may deadlock workflow instances, making Blocking Tasks for Tranquility generally unsafe—also for single updates. Vandewoude et al. [60] use Waiting for Tranquility and resort to Blocking Instances if the update point is not reached.

6.4 Comparing the Update Conditions

Figure 4 shows the execution of Algorithm 1 in the trip booking saga (Section 2.1) for the discussed update conditions. The columns show the time intervals from Figure 2 (aligning with Table 1). The rows show the update conditions. Each cell depicts the announcements 🔊, markings 📍, and locks 🔒 stored on both components at the given time interval when using the respective update condition. They further show whether a component can be updated safely, only for non-essential changes, or the update is unsafe.

Time interval A starts after BeforeWorkflow completes and ends before BeforeEachTask starts for the first time. All other time intervals are during the execution of the respective task.

As an example, the bottom row shows Essential Safety. The first entry illustrates that at time interval A, both the holiday and the car rental component hold an announcement for the workflow instance I executed on workflow engine r. Both components can be safely updated. In the second entry at time interval B, the holiday component is additionally marked and locked. Hence, it cannot be safely updated. At time interval C, the holiday component can be updated in case the update is a non-essential change because it holds no lock. However, an essential update would be unsafe because it holds a marking *and* an announcement of I.

Quiescence exhibits the fewest safe update intervals because it does not consider run time information on workflow instances. We assume that no new workflow instance is started after the one in Figure 2. Otherwise, there would be no safe update interval for Quiescence at all. In contrast, the safe update intervals for the other approaches remain unchanged without such an assumption.

Tranquility features all safe update intervals of Version Consistency. Plus, it permits intervals C and c for the holiday component (✓̸ in Table 1), which cannot be considered generally safe if the component does not respect the black-box principle. For instance, if the holiday component is updated at these intervals in a way that it writes and reads the hotel booking id in another format to/from workflows state S, "cancel hotel" could fail.

Essential Safety features all safe update intervals of Version Consistency for essential changes. For non-essential changes, it provides the most safe update intervals of all approaches.

Table 3: Construction statistics of the realistic collaborative BPMN workflows dataset based on RePROSitory [17].

Workflow Collection	Size	Recovered	Unchanged
RePROSitory	572		
Collaborative workflows	135		
Stuck	37	22	
Endless loop	9	6	
Incomplete	1	0	
Missing internals	10	0	
Evaluation dataset	106	28	78

7 EVALUATION

In this section, we empirically evaluate dynamic software updating for long-running and highly-frequent workflows. We aim to answer the following research questions.

RQ1: Can safe DSU be adopted in real-world collaborative workflow applications? With this question, we empirically investigate whether, with our model for safe DSU in workflows (Section 4.1), the approaches discussed in Section 4.2 and 6.2 can be applied to real-world collaborative workflow applications.

RQ2: Does Essential Safety significantly reduce the performance overhead of safe DSU in realistic workflows? This question empirically investigates the performance differences among the DSU approaches discussed analytically in Section 6.4. Specifically, we assess whether Essential Safety significantly reduces the impact of updating realistic workflow applications.

RQ3: How does the share of non-essential changes impact the performance of Essential Safety? This question investigates to which degree Essential Safety's performance depends on the amount of non-essential changes, estimating the share of non-essential changes that is sufficient to achieve better performance than the previous approaches. With RQ4, RQ3 validates the assumptions motivating the Essential Safety's performance improvement.

RQ4: How frequent are non-essential changes in software systems with multiple components? This question verifies the hypothesis behind Essential Safety—that most updates are non-essential—ensuring the generalizability of our results. Such evidence is required to determine the significance and applicability of our approach to real-world workflows.

7.1 Applicability of Safe DSU to Workflows

We now evaluate whether our model and the safe DSU approaches are applicable to workflows. For this, we constructed a dataset of real-world BPMN workflows and implemented a discrete event-based simulation for safe DSU in workflows using the dissemination algorithm (Section 5.1). We assess Essential Safety (Section 4) and the other safe DSU approaches (Section 6.2). The simulation and all scripts and data are publicly available [56].

No standard benchmark for realistic workflow models exists, possibly due to their complexity and business relevance [53, 54]. RePROSitory [17] is a database of realistic BPMN workflows. Based on a full copy from August 3, 2021, we constructed an evaluation dataset containing 106 collaborative BPMN workflows (Table 3). We selected all collaborative workflows with two or more BPMN lanes or pools—the BPMN elements that assign process elements to collaboration participants—which we interpret as workflow components. Everything outside any lane or pool is a separate component. Accordingly, all workflows have tasks on at least two different components. 57 workflows are not executable because they get stuck, include endless loops, are incomplete, or only contain the internal workflow of one lane or pool. We manually recovered 28 of these with minimal changes.[1]

Table 4: Simulation parameter distributions.

Parameter	Distribution
Network latency	Weibull: $\alpha = 1.5$, $\beta = 30\,\text{ms}$ ($\mu = 27.1\,\text{ms}$, sd $= 18.4\,\text{ms}$)
Instances per workflow	Weibull: $\alpha = 1.5$, $\beta = 20\,160$ ($\mu = 18\,199$, sd $= 12\,357$)
Avg. task duration	Weibull: $\alpha = 1.5$, $\beta = 2\,\text{min}$ ($\mu = 108.3\,\text{s}$, sd $= 73.6\,\text{s}$)
Task duration	Gaussian: sd $= 10\,\% * \mu$
Avg. update interval	Gaussian: $\mu = 12\,\text{h}$, sd $= 4\,\text{h}$, min $= 1\,\text{h}$, max $= 24\,\text{h}$
Avg. update duration	Weibull: $\alpha = 1.5$, $\beta = 5\,\text{min}$ ($\mu = 4.5\,\text{min}$, sd $= 3.1\,\text{min}$)
Update duration	Gaussian: sd $= 20\,\% * \mu$

Table 4 provides the simulation parameter distributions. All parameters are chosen with the intent to be as realistic as possible. Interarrival parameters and durations are Weibull-distributed with $\alpha = 1.5$, commonly used for Internet-based traffic simulations [5]. Update intervals and task durations are Gaussian-distributed, simulating regular CI/CD executions and tasks with predictable, roughly constant execution time, which is common in business applications. The workflow instances are distributed over ten workflow engines. For each workflow, the number of instances is Weibull-distributed with, on average, one invocation every 66 s. 99.7 % of the components are updated between once per day and once per hour, which are fixed limits of the mean update frequency. We draw the points in time for starting workflow instances and triggering component updates from a uniform distribution over the simulation timespan of two weeks. 90 % of the updates are non-essential changes. We performed a sensitivity analysis with both double and half the value of each parameter using the trip booking saga (Section 2.1). We omit the plots (reported in [56]), as they do only confirm obvious correlations, e.g., halving the task duration means increases updatability and decreases workflow duration and update time.

We successfully simulated all 106 workflows for the safe DSU approaches in Section 4.2 and 6.2 and the reaching strategies (Section 4.3). This result positively answers RQ1, showing that safe DSU can be applied to real-world collaborative workflow applications.

7.2 Performance of Essential Safety

We now investigate the performance differences of the DSU approaches in the simulation introduced before. Table 5 and Figure 5 compare the updating approaches (Section 4.2 and 6.2) and reaching strategies (Section 4.3) with the baseline "No Updates" where no updates are performed. All simulations are executed on the same trace of workflow instance executions and updates. *Updatability* is the overall time in which the update condition is met at the components (i.e., when updating is safe). The *update time* measures the timespan from triggering to completing a component update. It is split into the *update timeliness* (until the update condition is met) and the *update duration* (after the update condition is met). The

[1] All exclusions and adjustments are documented in the dataset's build script [56].

Table 5: Performance metric means for all update approaches and reaching strategies over all simulations in minutes.

Approach / Strategy	Updatability	Update		Workflow	
		Duration	Timeliness	Duration	Delay
No Updates (Baseline)				15.9	0.0
Quiescence	2 228.7	9.1	1 384.4	16.6	0.7
Version Consistency	4 510.5	8.2	518.0	16.7	0.8
Tranquility	902.7	8.1	184.7	15.5	0.5
Essential Safety (ES)	4 870.6	5.1	409.5	16.3	0.4
ES: essential changes	4 384.7	8.0	541.8		
ES: non-ess. changes	4 924.6	4.8	395.4		
Blocking Instances	4 219.3	4.3	11.4	17.4	1.5
Blocking Tasks (BT)	5 309.2	4.4	785.2	16.1	0.7
BT w/o Tranquility	4 651.5	4.4	6.3	16.7	0.8
Concurrent Versions	3 744.6	16.3	0.5	15.9	0.0
Waiting	4 330.0	4.3	2 402.3	16.1	0.2

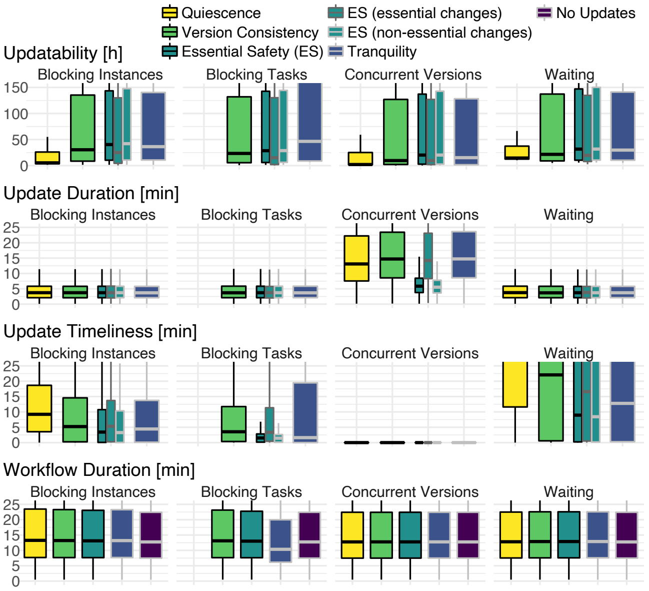


Figure 5: Performance of the safe DSU approaches.

workflow duration is the timespan between workflow instance start and completion. The *workflow delay* is the difference of the workflow instances' start to its start in the baseline. Analogously, the *workflow interruption* is the difference of the instances' completion times, measured by the sum of the instances' delay and duration differences. For Essential Safety, we also report the metrics separately for essential and non-essential changes, a distinction that does not lead to different results for the other approaches.

Among all approaches, we find the least performance impact on updates and workflow instances with Tranquility. However, Tranquility is generally unsafe—in contrast to all other approaches. Further, Tranquility with Blocking Tasks is the only simulation with deadlocked workflow instances (47.2 %), preventing their completion. These deadlocks positively skew the averages of all metrics for Tranquility[2] because deadlocked workflow instances are excluded from the measurement data. We observe that Essential Safety has similar performance to Version Consistency for essential changes and slightly better performance than Tranquility for non-essential changes. Overall, Essential Safety's performance is similar to Tranquility, but retains update safety. On average, Essential Safety's workflow interruption is 5.0 %, and it provides 8.0 % higher updatability, 21.2 % less update time, and 47.8 % less workflow interruption than Version Consistency—the best, safe competitor.

The reaching strategies' relative performance trends are similar among the updating approaches. All strategies add only a small delay to the workflows. Version Consistency exhibits no delay at all. Blocking Instances entails the highest delays. For Concurrent Versions, the update timeliness is similarly low for all updating approaches, whereas the update duration exhibits some variability. Vice versa, all other reaching strategies exhibit similar update durations but variable update timeliness. This difference is due to Concurrent Versions running two component versions in parallel, eliminating the workflow interruption because workflow instances never wait for component updates. However, Concurrent Versions requires that the components' implementations support running two different versions in parallel, which our simulation assumes is possible. If not supported by the components' implementations, Concurrent Versions must not be used. Instead, Blocking Tasks causes the least impact on component updates in such cases. Though Waiting delays workflow instances to a lesser extent, it heavily delays updates—on average by 40.0 hours.

Our results answer RQ2 and show that the impact of safe DSU can be significant. For Essential Safety, the impact of non-essential changes is completely negligible, and the impact of essential changes is not higher than with previous approaches. For a realistic change-set, Essential Safety significantly decreases the overhead of safe DSU compared to the state of the art.

7.3 Effect of Non-Essential Updates

To evaluate whether distinguishing essential and non-essential changes is effective—the assumption behind Essential Safety—we repeat the previous simulation with different ratios of non-essential to essential changes. Figure 6 shows the metrics from Section 7.2 for Essential Safety with only essential changes (0 %), only non-essential changes (100 %), and all ratios in between in steps of 20 % points. The results are presented separately for essential (ess.) and non-essential changes (non-ess.) as well as combined (total).

In total, the updatability increases with the share of non-essential changes; on average, 9.5 % from 0 % to 100 % non-essential changes. The update time reduces, on average, by up to 52.2 % and the workflow interruption by 54.8 %.

We now consider essential and non-essential changes separately. For all metrics and approaches, the results are similar, i.e., independent from the share of non-essential changes, except for the following cases: For Blocking Instances and Blocking Tasks, the updatabilities slightly decrease with the increasing share of non-essential changes because the likelihood that multiple updates are reached jointly and executed as a batch is lower. This increases blocking times to reach safe update intervals. For the same reason, the timeliness of essential change updates gets worse for Waiting.

[2] Due to the deadlock skew, Table 5 also reports Blocking Tasks without Tranquility.

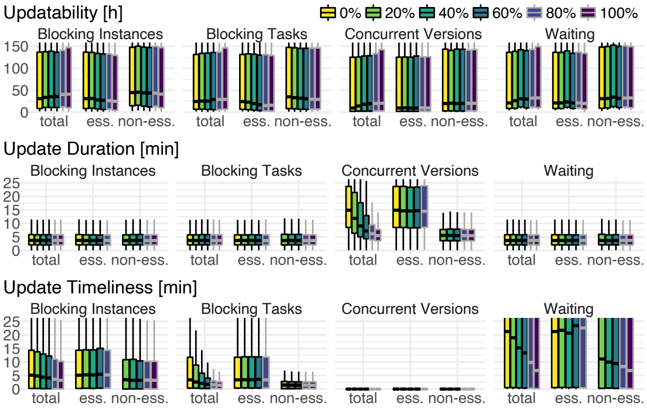


Figure 6: Performance of Essential Safety for different shares of non-essential changes (ess.: essential changes; non-ess.: non-essential changes; total: all changes).

The results answer RQ3: The higher the share of non-essential changes, the smaller Essential Safety's overall performance impact. Thus, considering non-essential changes is crucial and effective to reduce the overhead of safe DSU for workflows.

7.4 Frequency of Non-Essential Updates

To answer RQ4, we focus on open-source software repositories and assume that they use a continuous deployment pipeline. In continuous deployment, every commit may trigger the deployment of an update. Identifying which components are affected by each commit requires application knowledge and cannot be easily automated. Hence, for simplicity, a common practice is to redeploy all components, even though one can easily hypothesize that a subset suffices. To assess this hypothesis, we focus on repositories that aggregate various software components. Such "monorepos" are widely used [25, 33, 36, 46, 47]. Typically, the degree to which components in them depend on each other varies, allowing us to identify the subset of components that a commit changed.

We investigated eight monorepos that are publicly available on GitHub and described in [16]. In the monorepos, each component is encapsulated in its own directory. We identify the directories that contain a component based on the repository description. We explore the most recent 10 000 commits of each repository to determine how many components are affected by each commit. We assume that a component changed if a commit modifies a file in the component's directory. For commits that change files not associated to any component, we consider a conservative approximation (upper bound), that the commits affect every component, and a speculative approximation (lower bound), that they do not affect any. We ignore changes to tests, documentation, and hidden directories.

Table 6 shows for each monorepo the absolute number and the percentage of affected components (mean over all commits). On average, even under the conservative approximation, a commit affects less than half of the components. Under the speculative approximation, commits affect less than 10 % of the components in most monorepos. Accordingly, at least 60 % – 90 % of the component updates are non-essential changes.

Table 6: Affected components per commit in monorepos [16].

Monorepo	Average number of components affected by a commit	Average share of components affected by a commit
StartupOS	13	13 % – 41 %
Foursquare Fsq.io	13	8 % – 38 %
M3	22	8 % – 20 %
Celo	23	7 % – 10 %
Berty	31	8 % – 32 %
Stellar Go	41	3 % – 4 %
Habitat	49	5 % – 12 %
Nixpkgs	810	< 1 % – 15 %

So far we have demonstrated how often commits are non-essential component changes because the commits do not change a component's code. Additionally, not all code changes introduce semantic changes, i.e., they are non-essential for *all* components. Such commits further reduce the observed numbers in Table 6. Previous studies provide evidence that the amount of such non-essential changes is significant: (1) Kawrykow and Robillard [31] analyzed seven open-source Java systems, finding that up to 15.5 % of the method updates are cosmetic, behavior-preserving, or unlikely to provide further insight into component relationships. (2) Based on the TravisTorrent dataset [9], Abdalkareem et al. [2] found that 10 % of the commits developers manually skip in CI/CD pipelines are skipped because they are non-essential changes, i.e., they only touch documentation, source code comments, formatting of source code, meta files, or are code release preparations.

These results answer RQ4: On average, 60 % of the component changes are non-essential as a lower bound, while we realistically assume a considerably higher percentage of over 90 %.

8 CONCLUSION

Traditionally, software updates require shutting down the system before replacing any component. To avoid service disruption, *dynamic software updating* (DSU) techniques ensure that components can safely be replaced while the application is running. Unfortunately, existing safe DSU approaches introduce a significant performance overhead, and it is unclear how to apply them to *workflows*, i.e., long-running, asynchronous transactions. To close this gap, we propose a unified formal model for safe DSU in workflows, show how state-of-the-art DSU approaches are captured by it, and compare them analytically with Essential Safety, our novel safe DSU solution. Essential Safety leverages the identification of updates that have no semantic changes—non-essential changes—effectively reducing the performance overhead of DSU. The empirical evaluation on 106 realistic collaborative BPMN workflows and eight monorepos confirms that we enable efficient, safe dynamic software updating in long-running and frequently executed workflows.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and all students at the Technical University of Darmstadt who contributed to the discussion on safe DSU. This work has been co-funded by the Swiss National Science Foundation (SNSF, No. 200429), by the German Research Foundation (DFG, SFB 1119), by the Hessian LOEWE initiative (emergenCITY and Software-Factory 4.0), and by the University of St. Gallen (IPF, No. 1031569).

REFERENCES

[1] 2021. Kubernetes. https://kubernetes.io/, last accessed on 2021-10-22.

[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2021. Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering* 47, 3 (March 2021), 448–463. https://doi.org/10.1109/TSE.2019.2897300

[3] Amazon Web Services. 2021. Amazon States Language. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html, last accessed on 2021-07-30.

[4] Amazon Web Services. 2021. AWS Step Functions. https://aws.amazon.com/step-functions, last accessed on 2021-08-20.

[5] Muhammad Asad Arfeen, Krys Pawlikowski, Don McNickle, and Andreas Willig. 2013. The role of the Weibull distribution in Internet traffic modeling. In *Proceedings of the 2013 25th International Teletraffic Congress (ITC)*. 1–8. https://doi.org/10.1109/ITC.2013.6662948

[6] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910

[7] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. *Serverless Computing: Current Trends and Open Problems*. Springer, Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

[8] Luciano Baresi, Carlo Ghezzi, Xiaoxing Ma, and Valerio Panzica La Manna. 2017. Efficient Dynamic Updates of Distributed Components Through Version Consistency. *IEEE Transactions on Software Engineering* 43, 4 (2017), 340–358. https://doi.org/10.1109/TSE.2016.2592913

[9] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, 447–450. https://doi.org/10.1109/MSR.2017.24

[10] Boutheina Bennour, Ludovic Henrio, and Marcela Rivera. 2009. A Reconfiguration Framework for Distributed Components. In *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime* (Amsterdam, The Netherlands) *(SINTER '09)*. Association for Computing Machinery, New York, NY, USA, 49–56. https://doi.org/10.1145/1596495.1596509

[11] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos Zarras. 1998. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159) (CDS '98)*. IEEE Computer Society, USA, 35–42. https://doi.org/10.1109/CDS.1998.675756

[12] Camunda. 2016. Zeebe – Workflow Engine for Microservices Orchestration. https://zeebe.io/, last accessed on 2021-09-02.

[13] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. 1998. Workflow evolution. *Data & Knowledge Engineering* 24, 3 (1998), 211–238. https://doi.org/10.1016/S0169-023X(97)00033-5 ER '96.

[14] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. 2015. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology* 57 (2015), 21–31. https://doi.org/10.1016/j.infsof.2014.07.009

[15] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. 2015. On the Journey to Continuous Deployment: Technical and Social Challenges along the Way. *Information and Software Technology* 57 (2015), 21–31. https://doi.org/10.1016/j.infsof.2014.07.009

[16] Uriel Corfa. 2017. Awesome Monorepo: Notable public monorepos. https://github.com/korfuri/awesome-monorepo#notable-public-monorepos, last accessed on 2021-09-02.

[17] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, and Francesco Tiezzi. 2019. RePROSitory: a Repository Platform for Sharing Business PROcess modelS. *BPM (PhD/Demos)* (2019), 149–153.

[18] R. S. Fabry. 1976. How to Design a System in Which Modules Can Be Changed on the Fly. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, California, USA) *(ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 470–476.

[19] Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. 2019. *2019 Accelerate State of DevOps Report*. Technical Report. https://services.google.com/fh/files/misc/state-of-devops-2019.pdf, last accessed on 2021-05-05.

[20] Martin Fowler. 2010. Blue Green Deployment. https://martinfowler.com/bliki/BlueGreenDeployment.html, last accessed on 2021-07-22.

[21] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. 2020. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3048335

[22] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 249–259. https://doi.org/10.1145/38713.38742

[23] Matthias Geiger, Simon Harrer, Jörg Lenhard, and Guido Wirtz. 2016. On the Evolution of BPMN 2.0 Support and Implementation. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE '16)*. 101–110. https://doi.org/10.1109/SOSE.2016.39

[24] Matthias Geiger, Simon Harrer, Jörg Lenhard, and Guido Wirtz. 2018. BPMN 2.0: The state of support and implementation. *Future Generation Computer Systems* 80 (2018), 250–262. https://doi.org/10.1016/j.future.2017.01.006

[25] Durham Goode and Rain. 2014. Scaling Mercurial at Facebook. https://engineering.fb.com/2014/01/07/core-data/scaling-mercurial-at-facebook/, last accessed on 2021-09-02.

[26] Google Cloud. 2021. Workflows. https://cloud.google.com/workflows, last accessed on 2021-08-20.

[27] Paul Grefen, Jochem Vonk, and Peter Apers. 2001. Global Transaction Support for Workflow Management Systems: From Formal Specification to Practical Implementation. *The VLDB Journal* 10, 4 (Dec. 2001), 316–333. https://doi.org/10.1007/s007780100056

[28] Tianxiao Gu, Xiaoxing Ma, Chang Xu, Yanyan Jiang, Chun Cao, and Jian Lu. 2018. Automating Object Transformations for Dynamic Software Updating via Online Execution Synthesis. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.19

[29] Deepak Gupta, Pankaj Jalote, and Gautam Barua. 1996. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering* 22, 2 (1996), 120–131. https://doi.org/10.1109/32.485222

[30] Michael W. Hicks and Scott Nettles. 2005. Dynamic software updating. *ACM Trans. Program. Lang. Syst.* 27, 6 (2005), 1049–1096. https://doi.org/10.1145/1108970.1108971

[31] David Kawrykow and Martin P. Robillard. 2011. Non-Essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/1985793.1985842

[32] Jeff Kramer and Jeff Magee. 1990. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering* 16, 11 (1990), 1293–1306. https://doi.org/10.1109/32.60317

[33] Frederic Lardinois. 2017. Microsoft now uses Git and GVFS to develop Windows. https://techcrunch.com/2017/05/24/microsoft-now-uses-git-and-gvfs-to-develop-windows/, last accessed on 2021-09-02.

[34] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. 2017. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology* 82 (2017), 55–79. https://doi.org/10.1016/j.infsof.2016.10.001

[35] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, last accessed on 2021-09-02.

[36] Aimee Lucido. 2017. Monorepo to Multirepo and Back Again. Presentation at the Uber Technology Day, https://www.youtube.com/watch?v=lV8-1S28ycM, last accessed on 2021-09-02.

[37] Lucy Ellen Lwakatare, Terhi Kilamo, Teemu Karvonen, Tanja Sauvola, Ville Heikkilä, Juha Itkonen, Pasi Kuvaja, Tommi Mikkonen, Markku Oivo, and Casper Lassenius. 2019. DevOps in practice: A multiple case study of five companies. *Information and Software Technology* 114 (2019), 217–230. https://doi.org/10.1016/j.infsof.2019.06.010

[38] Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. 2011. Version-Consistent Dynamic Reconfiguration of Component-Based Distributed Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 245–255. https://doi.org/10.1145/2025113.2025148

[39] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. 2004. Composing adaptive software. *Computer* 37, 7 (2004), 56–64. https://doi.org/10.1109/MC.2004.48

[40] Frederic Montagut, Refik Molva, and Silvan Tecumseh Golega. 2008. The Pervasive Workflow: A Decentralized Workflow System Supporting Long-Running Transactions. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38, 3 (2008), 319–333. https://doi.org/10.1109/TSMCC.2008.919184

[41] Kief Morris. 2021. *Infrastructure as Code: Dynamic Systems for the Cloud Age* (second ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.

[42] Netflix. 2021. Conductor. https://netflix.github.io/conductor/, last accessed on 2021-08-20.

[43] Sam Newman. 2021. *Building Microservices* (second ed.). O'Reilly Media, Inc.

[44] OASIS. 2007. Web Services Business Process Execution Language Version 2.0. OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, last accessed on 2021-07-30.

[45] Object Management Group. 2014. Business Process Model and Notation Version 2.0.2. https://www.omg.org/spec/BPMN, last accessed on 2021-07-30.

[46] Dorothy Ordogh. 2018. Pants and Monorepos. Presentation at the Typelevel Summit Boston, https://www.youtube.com/watch?v=IL6LBWNi3fE, last accessed on 2021-09-02.

[47] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (June 2016), 78–87. https://doi.org/10.1145/2854146

[48] Bernd Rücker. 2019. trip-booking-saga-serverless. GitHub repository, https://github.com/berndruecker/trip-booking-saga-serverless, last accessed on 2021-07-27.

[49] Marcio Augusto Sekeff Sallem and Francisco Jose da Silva e Silva. 2006. Adapta: A Framework for Dynamic Reconfiguration of Distributed Applications. In *Proceedings of the 5th Workshop on Adaptive and Reflective Middleware (ARM '06)* (Melbourne, Australia) *(ARM '06)*. Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/1175855.1175865

[50] Danilo Sato. 2014. Carnary Release. https://martinfowler.com/bliki/CanaryRelease.html, last accessed on 2021-07-22.

[51] Danilo Sato. 2014. Parallel Change. https://martinfowler.com/bliki/ParallelChange.html, last accessed on 2021-07-22.

[52] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. 2013. A survey of dynamic software updating. *Journal of Software: Evolution and Process* 25, 5 (2013), 535–568. https://doi.org/10.1002/smr.1556 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1556

[53] Marigianna Skouradaki, Vasilios Andrikopoulos, and Frank Leymann. 2016. Representative BPMN 2.0 Process Model Generation from Recurring Structures. In *2016 IEEE International Conference on Web Services (ICWS)*. 468–475. https://doi.org/10.1109/ICWS.2016.67

[54] Marigianna Skouradaki, Dieter H. Roller, Frank Leymann, Vincenzo Ferme, and Cesare Pautasso. 2015. On the Road to Benchmarking BPMN 2.0 Workflow Engines. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 301–304. https://doi.org/10.1145/2668930.2695527

[55] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. Automating Serverless Deployments for DevOps Organizations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 57–69. https://doi.org/10.1145/3468264.3468575

[56] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2022. Evaluation of Safe Dynamic Updating on Collaborative BPMN Workflows With a Discrete-event Simulation: Dataset, Implementation, Measurements, and Analysis. https://doi.org/10.5281/zenodo.5864684

[57] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. 2007. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.* 29, 4 (Aug. 2007), 22–es. https://doi.org/10.1145/1255450.1255455

[58] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. 2009. Dynamic Software Updates: A VM-Centric Approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1542476.1542478

[59] Ken Tindell. 1990. Dynamic Code Replacement and Ada. *Ada Lett.* X, 7 (Aug. 1990), 47–54. https://doi.org/10.1145/101120.101133

[60] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. 2007. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering* 33, 12 (2007), 856–868. https://doi.org/10.1109/TSE.2007.70733

[61] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. 2010. Dynamic Code Evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java* (Vienna, Austria) *(PPPJ '10)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/1852761.1852764