Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Parent Restarting for Scientific Workflow Systems

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (Ba. Sc.)

eingereicht von:	Matteo Koczorek
geboren am:	24.06.1990
geboren in:	Essen
Gutachter/innen:	Prof. Dr. rer. nat. Lars Grunske
	Prof. Dr. Ulf Leser

eingereicht am: .... verteidigt am: ....

Contents

1 Abstract

Scientific workflow management systems enable scientists to design and execute large-scale, multi-step data processing by leveraging parallel- and cloud computing. These systems process data using directed acyclic topologies. A workflow is formed as a directed acyclic graph of individual nodes, that each execute a specific task. Like any piece of software, the code that implements these workflows can be flawed. Existing research has produced and evaluated a wide array of automatic error recovery strategies to automatically recover from bugs encountered at runtime, in order to avoid expensive re-executions. In some cases, bugs can be propagated from parent nodes to child nodes. In these cases, the node that surfaces an error is not the source of the error, but merely the node that could not propagate the error any further. This specific scenario of propagated bugs is not covered by the existing solutions and thus poses a research gap.

The objective of this thesis is to implement and evaluate a novel error recovery strategy that specifically targets propagated bugs, by restarting the parent nodes of nodes that produce errors at runtime.

The novel error recovery strategy is implemented by forking the popular scientific workflow management system Nextflow. The fork is then evaluated by benchmarking it on a set of workflows against the public version of Nextflow.

The results show, that the solution is effective and more efficient compared to the baseline for workflows that are specifically manipulated to exhibit the behavior targeted by this work. Due to the limitations of the research, there remain some threats to external and internal validity, and thus it remains to be seen if the efficiency of the solution observed in the evaluation translates to the real world. The results, however, indicate that the solution is promising and, therefore, this work suggests further evaluation of the parent retry error recovery strategy.

2 Introduction

Scientific workflow management systems are software tools that enable scientists to design and execute complex, multi-step computational tasks such as data analysis, experiment evaluation, or simulation. Through an abstraction layer, scientists are enabled to leverage distributed and cloud computing to perform tasks with high computational effort in a time-efficient manner. The systems often depend on scripting languages such as Python, Groovy, or Bash as a means for users to design and configure their workflows. Still, they may also provide abstract graphical user interfaces to do so. These systems process data using an acyclic topology. These topologies consist of operators which pass data around. The acyclic nature of the topologies enables workflow management systems to leverage parallelization and cloud computing to increase efficiency and performance.

Like any software, the workflows executed by these systems are subject to unexpected and undesired behavior due to flawed software code. The use cases for scientific workflow management systems usually revolve around large experiments and long-running data analysis. Dismissing the results of a workflow after encountering a bug requires a complete and expensive re-execution. Since executions tend to be time-consuming and resource intensive, this is especially undesirable. In order to avoid that there have been extensive efforts to allow for the completion of a workflow despite the appearance of unexpected behavior 3.2. The existing research and development have produced a wide array of automated error recovery strategies that strive to recover from bugs in the nodes in which they occur. Existing strategies leave a gap of bugs that surface in a given node but originate from a parent node and are merely propagated. The symptomatic node in these cases is not the source of the bug but merely the node that could not propagate the bug further.

This thesis aims to design, implement and evaluate a novel error recovery strategy. When encountering an error in a given node the novel strategy attempts to restart not just the node itself, but also the node's parents, in order to recover from propagated bugs. The novel strategy will be integrated as a fork of the popular open-source framework Nextflow 3.4. Using this modified variant of Nextflow, the thesis aims to answer the following **research questions**:

RQ1. Is it possible and feasible to implement an error recovery strategy that restarts parent nodes of repeatedly failing nodes in the Nextflow code base?

RQ2. Is the restarting of parent nodes of repeatedly failing nodes a viable strategy to circumvent bugs that are propagated from parent nodes?

RQ1 questions whether it is possible to produce a functional solution that successfully circumvents the class of propagated bugs that are relevant to this work. If this work is able to produce such a solution and beyond that, is able to demonstrate that the solution is indeed functional, then the possibility of such a solution is demonstrated via proof of concept and the feasiblity is demonstrated due to the fact that the implemetation was possible in the scope of a bachelors thesis.

RQ2 questions the efficiency of the solution and through that its usefulness. Viability is demonstrated if the solution is able to successfully complete the workflow execution faster than the baseline of the only existing solution, which is to restart the entire workflow.

3 Background

3.1 Error Recovery in Scientific Workflow Management Systems

Scientific workflow management systems are typically used when dealing with high computational effort. The workflows therefore may be long-running and expensive.

Since the workflows execute arbitrary code written by users, just like any other piece of software, they may be flawed and contain bugs. Due to the nature of these workflows (i.e. high computational effort and long-running), it is desirable to recover from errors without the need to restart the entire workflow.

Error recovery strategies can be applied on different levels. The two levels that the methods are generally applied to are at task-level and at workflow-level.

3.1.1 Levels

Task-Level As the name suggests, task-level error recovery considers the task in isolation and deploys recovery attempts on that task without interacting with or modifying surrounding nodes or other parts of the workflow. An example of a task-level error recovery strategy is to retry a failing task until either the task succeeds or some threshold of maximum retries is reached. Task-level error recovery strategies are often masking and do not require human intervention or additional configuration. They are performed automatically in an attempt to continue the workflow without interruption.

Workflow-Level As opposed to task-level error recovery, workflow-level error recovery is holistic. It considers not only the failing task but also the rest of the workflow, such as surrounding nodes. These strategies tend to require significant additional input and configuration as they often require a semantic understanding of the context. One example would be alternative tasks which are provided by the user to substitute failing tasks. This effectively mutates the workflow and is, thus, at workflow level and because it requires the user to provide the alternative task, it cannot be produced by the system autonomously.

3.1.2 Forms of Fault Tolerance

Error recovery strategies under the umbrella of fault tolerance can be categorized into different forms of fault tolerance. Felix C. Gärtner proposed a formalization of fault tolerance [1] which is showcased in figure 1.

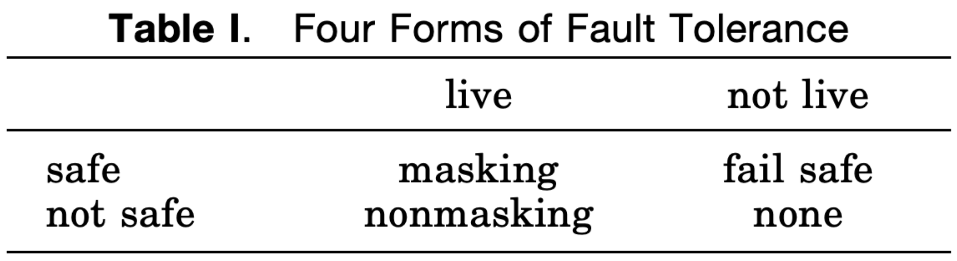
Table I. Four Forms of Fault Tolerance

	live	not live
safe	masking	fail safe
not safe	nonmasking	none

Figure 1: Table collecting four different possible forms of fault tolerance proposed by Felix C. Gärtner [1]

The two axes of figure 1 refer to the two properties of liveness and safety. Liveness is satisfied if a system eventually reaches an expected goal (”good thing” will happen[1]). Safety is satisfied if, at any point during the execution, no illegal state is reached (”bad thing” will not happen[1]). The form of fault tolerance ”none” where ”not safe” intersects with ”not live” should not be considered a form of fault tolerance according to the author as it means the absence of fault tolerance for a given fault class and is thereby not considered in the following. The form of fault tolerance ”fail safe” where ”safe” and ”not live” intersect, refers to a type of fault tolerance where the transition to an illegal state is prevented by interrupting the execution without eventually reaching the initial goal of the execution is relevant in safety-critical operations. As the goal of workflow system execution is to eventually reach a result and any failure of the system should not lead to safety issues this form is also not considered in the following. For the context of scientific workflow management systems, thereby, the two forms of fault tolerance ”masking” and ”non-masking” are left.

Masking Masking error recovery strategies both lead the system to the desired goal (liveness) and do so without transitioning into an interrupted state (safety) [1]. In the context of scientific workflow management systems masking error recovery strategies attempt to recover from an error automatically without any human intervention. This is achieved at runtime through various strategies such as retrying failing nodes or moving forward with preconfigured default values without interrupting the workflow.

Non-Masking Non-masking error recovery strategies lead the system to the desired goal (liveness) but may transition into an interrupted state while doing so (safety) [1]. In scientific workflow management systems, this interrupted state usually requires human intervention to return to a safe state.

3.2 Related Work

The subject of error recovery and fault tolerance in scientific workflow management systems has been studied and there are numerous publications about the topic [1, 2, 3, 4, 5, 6, 7, 8] The different publications propose and implement a wide array of error recovery strategies described in the following.

Specifically, Jia Yu and Rajkumar Buyya [4] lay out a categorized enumeration of different error recovery strategies.

3.2.1 Task-Level

Simple Retrying This error recovery strategy is considered the simplest error recovery strategy [4]. It is supported by most of the popular scientific workflow management systems. If configured, when a task throws an unhandled exception of any kind during execution, it is retried in an attempt to circumvent the issue that led to the initial crash. Depending on the workflow management system, this may be a global default strategy or configured on a per-node basis. Nextflow allows both a global configuration to use this method for every node, as well as a configuration on a per-node basis.

The process is usually repeated n times, where n is a user-configured value that describes the maximum number of retries.

Replication In this strategy, multiple replicas of a task are executed on different nodes simultaneously [5, 4]. Replication follows the same rationale as simple retrying: multiple attempts at execution may circumvent bugs that do not appear consistently. The difference here is that the task is executed multiple times in parallel instead of repeatedly upon error. Once one of the nodes succeeds that task is considered successful even if another node fails. This strategy reduces execution time in the worst case at the price of increased average compute resources. Depending on the use case, the complexity of the task, the available time, and the available resources either **simple retrying** or **replication** may be more desirable to the user.

Checkpointing Checkpointing dates back to before the conception of the first scientific workflow management and grid systems and has been evaluated for fault tolerance in computing systems[6].

Checkpointing in the context of scientific workflow management systems similarly closely relates to simple retrying but does not require the task to be retried from the beginning and instead allows for the task to be retried from a certain ”checkpoint” that is deemed safe. There has been significant effort in developing and evaluating tools that leverage checkpointing for error recovery in scientific workflow management and grid systems. Some systems, such as Pegasus or Apache Spark, support checkpointing out of the box [8, 9] but there are also frameworks such as Dome [7] or Fail-safe PVM[10] that offer tools for checkpointing.

Migration Migration is a strategy proposed by Gopi Kandaswamy and Anirban Mandal and Daniel A. Reed [11]. It refers to the transferal of an execution of a task to another compute node in an effort to circumvent issues that arise from the computational environment such as hardware issues, unfitting resources, or misconfigurations. It can be performed automatically and may be effective in environments that are prone to hardware and software issues of the environment.

3.2.2 Workflow-Level

Alternative Task With this strategy in case a task has failed, a substitution task is executed. This is possible when there are different implementations to achieve the goal of the original task. The alternative tasks must be provided by the user [4]. This beheavior can be applied if there are multiple implementations for a given task. For example, if one of the implementations is significantly more efficient but less reliable it can be performed optimistically as a first attempt and in case of failure, be substituted by a less efficient, but more reliable implementation[3]. The workflow is thereby altered at runtime, as one of the nodes is substituted with another.

Workflow-level redundancy Workflow-level redundancy is to **alternative task** what **replication** is to **simple retrying**. Just like the **alternative task** strategy, workflow-level redundancy relies on the existence of multiple different processes that perform the same task in different ways. Instead of one being the fallback for another however, in this strategy, all different variations are executed in parallel, and as soon as one of them completes successfully its result is considered the result of the entire task and the other may be terminated. Hwang and Kesselmann incorporate workflow-level redundancy as part of their flexible framework for fault tolerance [3].

User-defined Exception Handling This error handling allows the user to configure custom error handling behavior for specific tasks or specific errors [4]. This can be useful in case there exist expected specific exceptions where the user might already know a specific strategy curated to resolve the specific issue. One goal of this approach is to separate error recovery logic from application logic.

3.3 Research Gap

As demonstrated in section 3.2 there has been significant effort in the scientific community to design and evaluate a wide array of error recovery strategies for scientific workflow management systems. There appears to be a type of bug however, that is not yet covered by existing strategies. A problem the topologies of scientific workflow management systems can have is that bugs can propagate before surfacing. All mitigation strategies described above assume that the bug occurred in the node that failed to execute successfully. However, the task that fails might not be the task that generated the bug, just the task that could not propagate the bug. An example scenario is that a parent node produces incorrect output but does not throw a run-time exception. Instead, the task completes successfully, as far as the system is concerned, and passes its output to the input of a child. This child may now fail due to incorrect input, but because it was not the source of the bug, any attempt to retry the node or even fall back to a different implementation of the node would consistently produce the error as the bug has already occurred upstream in the DAG.

The goal of this work is to implement and evaluate a novel error recovery strategy that aims to mitigate propagated bugs in scientific workflow management systems. The basic idea is to restart the parent nodes of failing nodes where traditional error recovery strategies have failed to mitigate the issue. The strategy is evaluated by modifying the popular bioinformatics workflow management system nextflow.

3.4 Nextflow

Nextflow is a specific scientific worflow management system that is mainly used in bioinformatics for large data analyses such as genome analysis. Nextflow is an open-source software project released under the Apache 2.0 license. It is developed by Seqera Labs. Nextflow was initially released in 2014 and today is "one of the most widely used solutions for orchestrating scientific workflows".

Nextflow aims to make "techniques to analyze and run experiments on large datasets" available to the bioinformatics community while allowing the scientists to develop using their "favorite" programming language. "It supports deploying workflows on a variety of execution platforms including local, HPC schedulers, AWS Batch, Google Cloud Life Sciences, and Kubernetes. Additionally, it provides support for managing your workflow dependencies through built-in support for Conda, Spack, Docker, Podman, Singularity, Modules, and more."

The workflows themselves are defined as a set of processes that are then linked together into workflow definitions, which then form the directed acyclic execution graph or short DAG. Individual processes which form the nodes of the graph consist of input and output definitions as well as a script part that contains the actual computational instructions to perform the task for which the process is responsible. The scripts can be implemented using a variety of languages and tools such as Bash, Python, or Groovy.

3.5 Terminology

On an abstract level, scientific workflow management systems enable the execution of directed acyclic graphs (DAG). The nodes of these graphs are formed by individual units of work, often described as tasks. The workflow management system Nextflow specifically uses the keyword "process" to define such a task node in scripts.

3.6 Scope and Limitations

The scope of this work includes the implementation of a fork of Nextflow that exhibits the desired behavior of parent node retrying as an error recovery strategy and the evaluation of this fork on both synthetic workflows designed specifically to demonstrate this behavior as well as real-world workflows from the field of bioinformatics. Due to the very specific nature of these types of bugs, it is not realistic that such a bug will occur on a limited set of workflows that would fit into the scope of a bachelor thesis because of which the real-world workflows will be modified slightly to introduce such a bug. This is further explained and challenged as part of the research methodology and threats to validity section of this paper.

4 Approach

4.1 Challenges and motivation.

In order to achieve the novel error recovery strategy the Nextflow code needed to be modified in such a way, that the system supports an all-new error recovery strategy. This was achieved by forking the open-source repository of Nextflow and implementing the required changes.

Nextflow is built around the Groovy script parser and executes user-written code directly. The orchestration of the DAG is achieved by blocking inputs and outputs to individual processes in such a way, that those processes that can be parallelized, will be parallelized, and those that cannot will be executed sequentially. While this solution benefits from the basic components of Groovy, it does lead to an implicit execution of the DAG, which is not represented in the code and thus not easy to reason about and modify. The existing error recovery strategy of retrying individual process nodes is solved inside the class responsible for executing individual processes, because of that the behavior proposed in this document could not be implemented into the existing execution behavior of Nextflow in a straightforward manner.

4.2 Solution Design

As outlined above, the execution of the DAG is implemented implicitly, which made changing the order of execution and repeating already completed tasks at runtime difficult without a major refactoring and significantly changing how the system operates. In order to circumvent this issue the solution was implemented according to the flow presented in 2

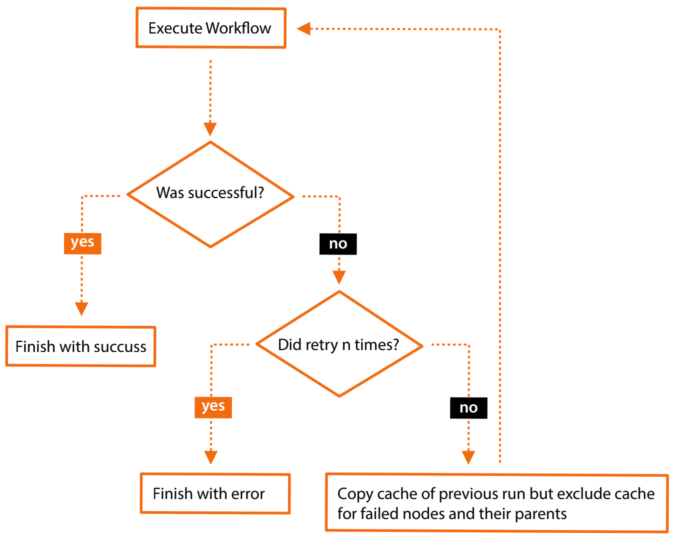

Figure 2: Implementing the desired behavior by injecting the previous runs cache into the next runs session, but excluding the failed processes and their parents, so that effectively the parents get retried as proposed by this document.

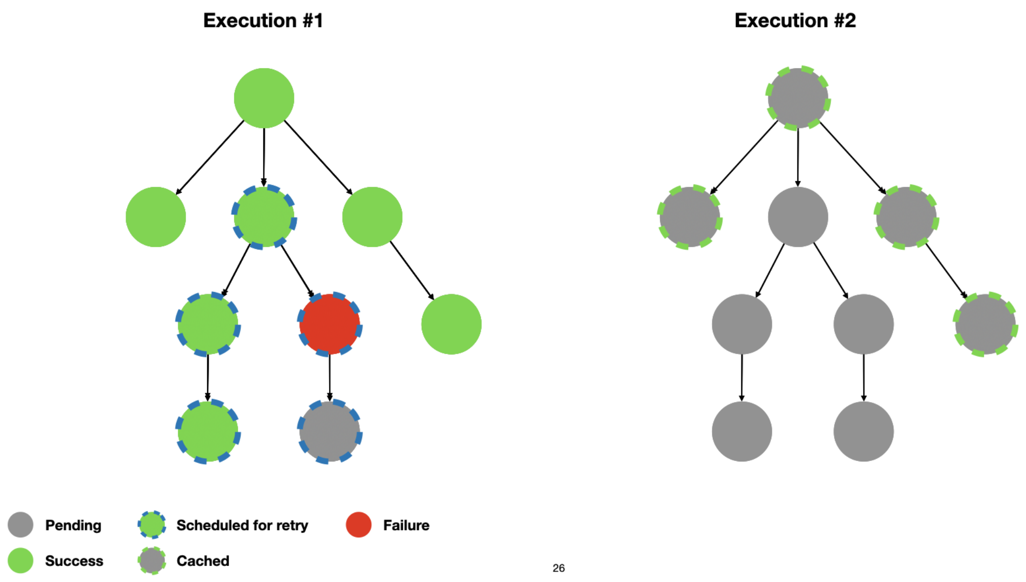

Figure 3: Restarting the entire workflow but injecting caches for the complement set of the set of those nodes that are scheduled for retry by the algorithm.

This results in two executions of the same run with caches injected for certain nodes.

4.3 Solution Implementation

As outlined above, the desired behavior was achieved implicitly. In reality, it is not just the parent nodes (and their descendants) that are being restarted, but instead the entire workflow itself. Effectively rerunning only the desired nodes is achieved by injecting caches from the previous run for the complement of the subset of nodes that are scheduled to be retried. In the subsequent run, the cached nodes will complete immediately without significant computational overhead and thus computational effort and execution time are only required for the nodes that need to be rerun, because they either did not finish in the previous run before the error occurred, or because they were identified to be rerun as part of the error recovery strategy outlined in this document.

4.4 Extraction of Hashing logic

The logic to generate a hash value that is used to store and retrieve the cache for a given process was included as part of the responsibilities of the TaskProcesser and implemented as a private method. Since the hashes for the tasks were needed to transfer caches from the session of a previous run to a subsequent run this hashing logic was extracted to a separate object that holds only this responsibility and can be injected into other compontents that need it.

```
interface ITaskHasher {
    HashCode createTaskHash(
        TaskRun task,
        UUID sessionId,
        CacheHelper.HashMode mode,
        Boolean isStubRun,
        Boolean shouldDumpHashes,
        ScriptBinding ownerScriptBinding,
        Map<String , Path> sessionBinEntries
    )
}
```

4.4.1 Graph Traversal Component to Identify the Family of a Node

The error recovery strategy aims to restart parent nodes of failing nodes. Restarting the parent nodes invalidates all results of all their descendants, including those that finished without exceptions. Nextflow collects a data structure named DAG that holds the entirety of the execution graph of the workflow. This data structure, however, contains not just a semantic representation of the execution graph but also contains the logic to produce it at runtime. Due to this responsibility, the object is complex and dependent on a lot of computational components of the code such as DataflowProcessor, TaskProcessor, TaskRun, and Channel as well as the actual input and output lists of the individual tasks. In order to enable the straightforward implementation and testing of search algorithms on the DAG object a mapper was implemented to transform the object to an isomorphic, but simpler representation of the DAG.

```
class EdgeRepresentation {
    VertexRepresentation from;
    VertexRepresentation to;
}
class VertexRepresentation {
    /**
    * The name of the process
    */
    String label;
    /**
    * The process object (here any simple object can be used while testing)
    */
    Object process;
}

class DAGRepresentation {
    /**
    * The list of edges in the graph
    */
    List<EdgeRepresentation> edges
    /**
    * The ordered list of vertices
    */
    List<VertexRepresentation> vertices
}
```

Using these data structures an algorithm could be implemented that takes a DagRepresentation and a VertextRepresentation and returns the set of the vertices' parent vertices and all their descendants which is referred to as the family of the vertex in this document and in the code base of this work.

```
interface IDAGFamilyFinder {
    /**
    * Returns the family for a given vertex. The family includes all parent
    * nodes of the vertex and all descendants of all parents (including the
    * vertex itself). Operators are skipped while traversing the graph.
    */
    Set<VertexRepresentation> findFamily(
        DAGRepresentation dag,
        VertexRepresentation vertex
    )
}
```

4.4.2 Integrating the Behaviour

The implementation of the components described above allowed for the integration of the behavior into Nextflow. The main method of the launcher was modified in a way that it inspects the session and all executed tasks after the execution is completed, and checks if any of the tasks were not successful. If any task was not successful and the maximum amount of parent retries is not yet reached, then a new launcher instance is created, the caches for a subset of processes are transferred to the Session of the new Launcher, and the new Launcher is once again executed.

```
class Launcher {

    ...

    static void main(String ... args) {
        final firstLauncher = new Launcher()
        runLauncher(firstLauncher, 0, args)
    }

    static void runLauncher(
        Launcher launcher,
        int numberOfRetries,
        String . . . args
    ){
        log.error("starting execution #" + (numberOfRetries + 1))
        final status = launcher.command(args).run()
        Session session = (launcher.command as CmdRun).runner.session;
        def hasFailedTask = session
            .taskExecutionTracker
            .allExecutedTaskRuns.findAll {
                t -> !t.isSuccess()
            }.size() > 0
        if(numberOfRetries < MAX NUMBER OF PARENT RETRIES && hasFailedTask) {
            final secondLauncher = new Launcher()
            transferCaches(launcher,secondLauncher)
            runLauncher(secondLauncher, numberOfRetries + 1, args)
        } else {
            System.exit(status)
        }
    }
}
```

These changes successfully implement the error recovery strategy outlined in this document as demonstrated by the evaluation.

5 Evaluation Design

The experiment aims to validate that the implementation works as expected and to answer the research questions posed by this work:

RQ1. Is it possible and feasible to implement an error recovery strategy that restarts parent nodes of repeatedly failing nodes in the Nextflow code base?

RQ2. Is the restarting of parent nodes of repeatedly failing nodes a viable strategy to circumvent bugs that are propagated from parent nodes?

In order to answer **RQ1** several workflows that are configured in such a way, that the novel error recovery strategy is invoked, are executed using the modified version. If the results show that these workflows have a higher success rate on the modified version of Nextflow than they have on the vanilla version then this question can be answered with yes, because this demonstrates that there exists a working proof of concept, proving that such a solution is possible and secondly, since this solution was developed in the scope of a bachelors thesis, that the implementation of such a solution is feasible.

In order to answer **RQ2** the solution needs to be benchmarked against an existing solution. Since no existing error recovery strategy is able to recover from this class of error (see 3.3), the benchmark to consider is restarting the entire workflow, as this would be the real word next step for unrecoverable failures during execution. If the results show that the novel solution performs on average faster than the benchmark of workflow-restarting, then the solution can be considered a viable strategy to circumvent bugs that are propagated from parent nodes.

5.1 Limitations

The best evaluation of the solution would be to run a large set of unmodified real-world open-source workflows without any interference or selection bias and to observe whether there is any significant difference in success rate and how the runtime and performance are overall affected. This is, however, not feasible due to various limitations. Firstly, the bugs that are relevant for this work are edge cases and, thus, the test set would have to be large enough so that these edge cases even occur. Secondly, real-world workflows are hard to come by. At the time of this writing, there is a total sum of 40 Nextflow workflows hosted on workflowhub.eu which likely would not be large enough to demonstrate the behavior. Thirdly, in addition to the rarity of the relevant types of bugs and the workflows themselves, these workflows are usually developed for the field of bioinformatics, because of which the setup and understanding requires deeper knowledge in this field which is out of scope for a computer science bachelor thesis. Finally the available workflows on workflowhub.eu often did not execute successfully in the testing and often missed related data due to dead links.

Because of these limitations the set of workflows, that the solution is tested on, consists of workflows that were either synthetically created or modified in such a way, that they produce the type of bug that is the subject of this work.

The synthetic nature of the test workflows has implications for the research questions. **RQ1** is less affected than **RQ2**. Even though the bugs are introduced artificially if the solution is able to circumvent those, it is reasonable to assume that the solution would be equally able to circumvent such bugs in a real-world context and thus effectiveness can be demonstrated despite the limitations. The limitations however will make the answer to **RQ2** less robust and subject to external threats to validity as efficient performance in a lab setting does not necessarily translate to efficient performance in the real world.

5.2 Workflows

In order to evaluate the solution with the limitations listed above in mind, three test workflows were collected. In total, there are two real-world workflows and one synthetically created workflow. The synthetic workflow was specifically designed to showcase the strength of the solution, while the real-world workflows were selected from a limited set of possibilities (see 5.1) and based on several desirable properties

1. The workflows should be representative of regular real-world Nextflow workflows. Because of that the workflows were picked from the field of bioinformatics.
2. The directed execution graphs of the workflows should contain enough nodes for the solution to be able to have any reasonable effect. For very small execution graphs the difference between restarting a parent node and restarting the entire workflow might be equivalent or almost equivalent and because of that evaluating the difference between the two would not be possible.

5.2.1 SARS-CoV-2 Genome Reconstruction

The first one is an open-source workflow for SARS-CoV-2 genome reconstruction via nanopore sequencing hosted on GitHub. A bug relevant to this solution was then artificially introduced into this workflow by making one of the processes produce a corrupted output file with a 50% chance.

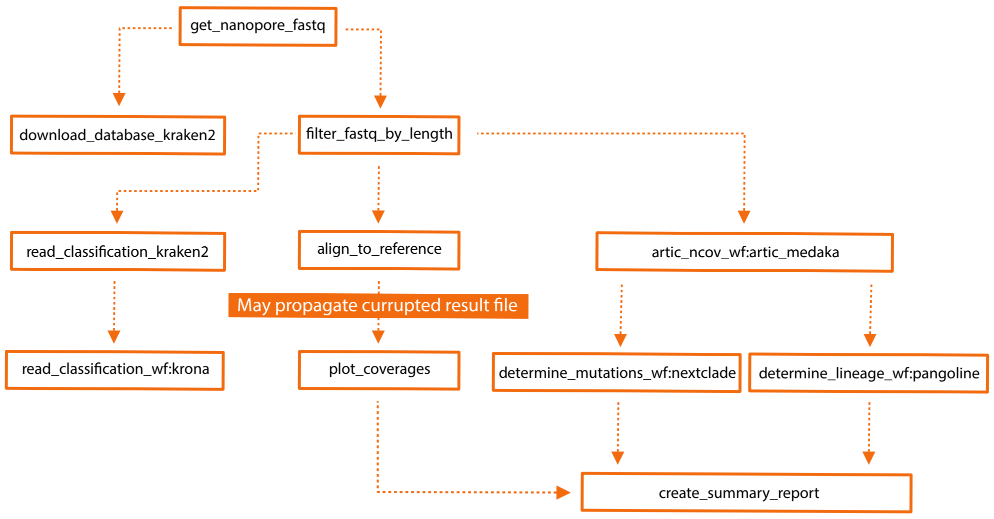
get_nanopore_fastq
download_database_kraken2
filter_fastq_by_length
read_classification_kraken2
align_to_reference
artic_ncov_wf:artic_medaka
May propagate currupted result file
read_classification_wf:krona
plot_coverages
determine_mutations_wf:nextclade
determine_lineage_wf:pangoline
create_summary_report

Figure 4: Simplified DAG of the workflow for "poreCov — SARS-CoV-2 Workflow for nanopore sequencing data" published by the Universitätsklinikum Jena. For the sake of displayability, several leaf nodes were removed. The process "align_to_reference" produces a corrupted output file with a 50% chance, which leads to "plot_coverages" failing.

5.2.2 Synthetic Workflow: Video Conversion

The second workflow for evaluation is a self-developed synthetic workflow that prepares a video file for upload to a fictional video hosting site. The workflow first converts the input video with the H264 video codec and then produces different videos with smaller resolutions and one with a vertical format from that. The smallest resolution video in addition is converted to a gif that functions as a mouseover preview for the fictional video platform.

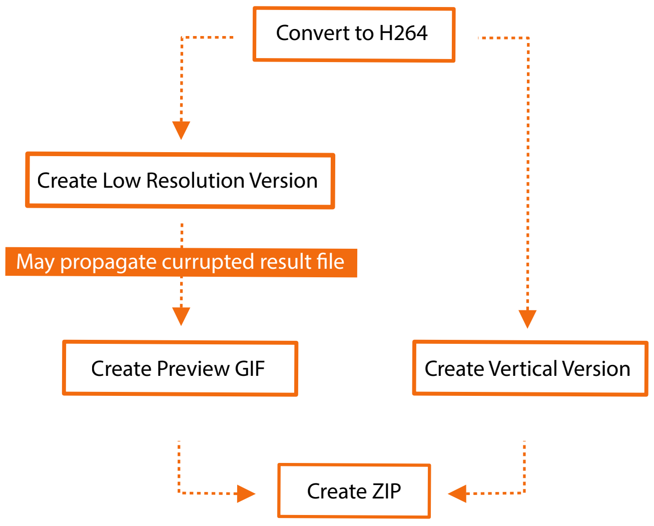

Figure 5: DAG of the synthetic "Video Conversion" workflow. The process "Create Low-Resolution Version" produces a corrupted output file with a 50% chance, which leads to "Create Preview GIF" failing.

5.2.3 Bacterial Genomes Workflow

The third workflow is an open source workflow for analyzing bacterial genomes hosted on GitHub. A bug relevant to this solution was then artificially introduced into this workflow by making one of the processes produce a corrupted output file with a 50% chance.

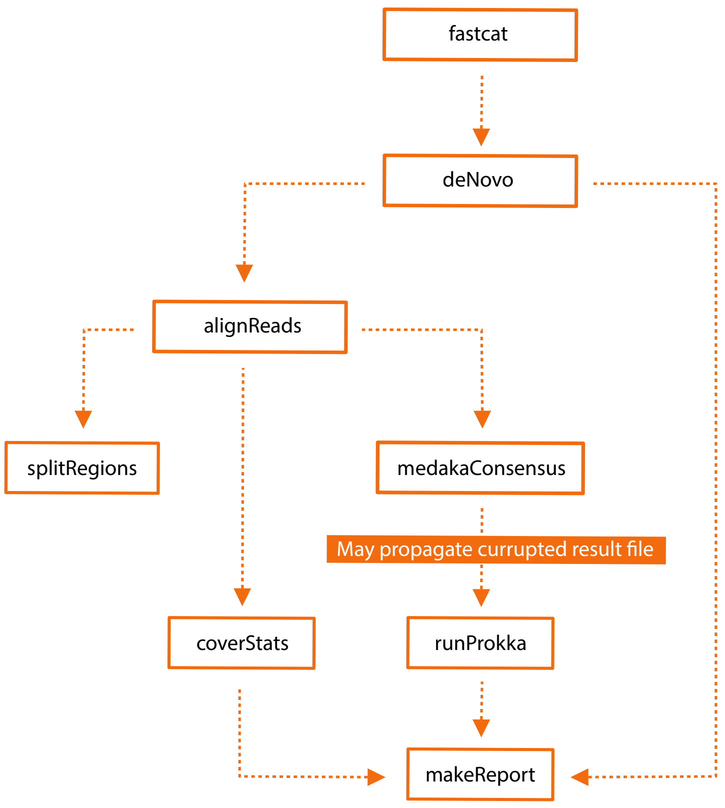

Figure 6: Simplified DAG of the workflow for "Bacterial Genomes Workflow" published by EPI2ME Labs. For the sake of displayability, several leaf nodes were removed. The process "medakaConsensus" produces a corrupted output file with a 50% chance, which leads to "runProkka" failing.

5.3 Experiment Setup

For the evaluation, the runtime performance of the modified version of Nextflow is compared against a baseline, which is defined as rerunning the entire workflow in an attempt to circumvent the bug as opposed to rerunning only parent nodes. Each of the different workflows is executed 200 times. At least 100 times on vanilla Nextflow and 100 times on the modified version of Nextflow.

The experiment setup is dockerized and executed in a cluster of virtual machines in a cloud. This enables large datasets for more accurate comparison, higher reliability due to the possibility of controlling the specifications precisely, and optimal reproducibility as anyone can repeat the experiment using any cloud provider.

1. CPU: 2,5 GHz Intel Core i7
2. RAM: 8 GB
3. OS: Ubuntu (Dockerized)

Each virtual machine continuously executes a single workflow on one of the two Nextflow versions and saves the resulting logs into an S3 bucket.

The logs are structured to allow for the collection of the data required to evaluate the solution. The logs contain the full output of Nextflow in addition to several necessary data logs such as start time, finish time, and start time of each subsequent run.

Using a Python script, a set of properties is then extracted from each log to represent the sample:

1. Start time of execution
2. Finish time of execution
3. Number of errors that occurred

This data can then be used to deduce the metrics that are taken into account for the evaluation:

1. Average duration overall
2. Average execution duration when no error occurred
3. Average execution duration when exactly one error occurred and the execution was retried once
4. Average execution duration when exactly two errors occurred and the execution was retried twice
5. Best-case performance
6. Worst-case performance

The resulting data should allow for the evaluation of effectiveness and efficiency and therefore allow for answering the research questions posed by this work. The script running vanilla Nextflow is configured to keep rerunning the entire workflow until success, to build the baseline against which to compare. This is not the case for the modified version of Nextflow, so the existence of workflow executions that completed successfully, despite errors being encountered, would already prove that the solution is capable of circumventing propagated bugs (**RQ1**). Since **RQ2** is not as easily demonstrated, the above-mentioned metrics mainly serve to evaluate **RQ2**. The most important metric is the average duration overall, which if improved, would demonstrate the efficiency of the solution (**RQ2**). The remaining metrics are collected to confirm certain aspects of the results. One such expectation is that the best-case performance should be unaffected, as the best cases would be those executions where no error occurred at all, and in these cases, both versions should be logically identical. Another expectation is that the worst-case performance should differ most as those would be the cases with the highest error count, and if the assumption is true, that the novel solution is faster at recovering from errors than the baseline, the overall difference in execution time should diverge further, the higher the error count.

6 Evaluation

The solution proposed in this document aims to deliver an automated error recovery strategy, in which specific nodes are rerun, while others are not, in an attempt to recover from an error that occurred at runtime in an effective and efficient manner. As explained in previous chapters, the specific nature of the type of error targeted by this solution requires either some modification of existing workflows or the generation of synthetic workflows in order to be able to showcase the functionality in the scope of a bachelor's thesis.

As the solution aims to recover from errors by retrying certain nodes, the baseline against which to evaluate is considered the automatic retrying of the entire workflow in regular Nextflow. The expectation here is that the two solutions are equally as effective, but the solution proposed in this document is more efficient as fewer nodes will have to be retried on average.

In regard to the **research questions**, for answering the **RQ1** it is important to observe if the modified version of Nextflow is able to consistently complete workflows, despite the occurrence of errors and thus demonstrating that it is indeed possible and feasible to develop a solution for this class of errors. For answering **RQ2** the performance of the novel solution compared to the baseline matters. For this, the average duration until success is collected and compared against the duration until success of the baseline.

Looking at the two sets of workflows W = {Genome Reconstruction, Video Conversion, Bacterial Genomes} and Nextflow versions V = {Vanilla Nextflow, Modified Nextflow} the total evaluation consists of the cross product of these two sets.

For each tuple of (workflow, Nextflow version) the workflow is executed on the respective Nextflow version 20 times while logging the runtime, whether the run was ultimately successful, and how many retries were attempted. For the sake of comparison, those runs that were ultimately not successful are removed from the set.

6.1 Video Conversion

The video conversion workflow is entirely synthetic and solely designed to demonstrate the functionality of the parent retry solution. The conditions of this workflow should be ideal.

Average Dur.	**Vanilla Nextflow**	**Modified Nextflow**	**Improvement**
Overall	22.24m	16.88m	24.11%
No Errors	17.05m	16.85m	1.18%
One Error	29.65m	16.86m	43.14%
Two Errors	33.33m	17.05m	48.84%

Figure 7: Table collecting the averages of 230 executions of the Video Conversion workflow with 115 executions performed on vanilla Nextflow and repeating until success when encountering an error, and 115 times on the modified Nextflow version using the novel error recovery strategy.

Looking at the results we can clearly see that the parent retry strategy significantly outperforms the baseline. In the happy case of initial success, the two solutions do not differ significantly, which is expected as there should be no difference in the execution logic in this case.

Variance	**Vanilla Nextflow**	**Modified Nextflow**	**Improvement**
Overall	52.54m	1.67m	96.82%
No Errors	5.04m	1.8m	64.22%
One Error	12.68m	1.52m	88.04%
Two Errors	0.0m	1.34m	-

Figure 8: Table collecting the variance of 230 executions of the Video Conversion workflow with 115 executions performed on vanilla Nextflow and repeating until success when encountering an error and 115 times on the modified Nextflow version using the novel error recovery strategy.

As evidenced by both figure 9 and figure 8, the variance is greatly reduced in the modified version due to the greatly reduced costs of additional runs compared to the baseline.

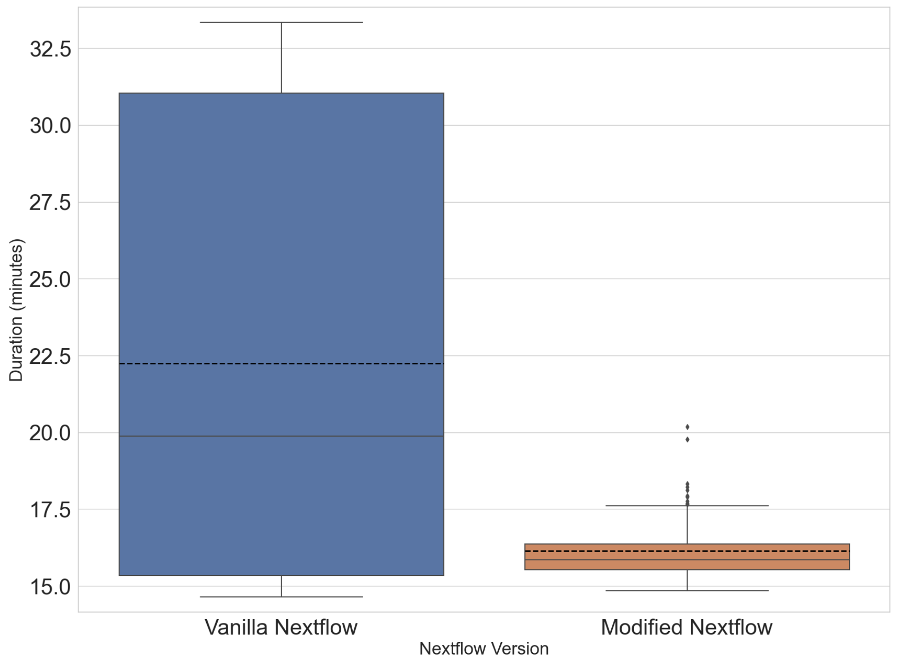

Figure 9: Boxplot collecting the results of 230 executions of the Video Conversion workflow with 115 executions performed on vanilla Nextflow and repeating until success when encountering an error and 115 times on the modified Nextflow version using the novel error recovery strategy.

The boxplot (see figure 9) further illustrates the difference between the results. The upper whiskers representing the worst cases with 2 errors encountered are more than twice as high for the baseline as they are for the modified version of Nextflow. This is in line with the almost doubled average duration in case of two errors for the vanilla version (see figure 7).

Overall the results show that in the synthetic workflow, the strategy proposed by this work is able to recover from the failures produced by the workflow and successfully finish executing the workflow without any human intervention and without the need to restart the entire workflow (**RQ1**). It is also demonstrated that the solution can be significantly more efficient compared to the status quo in recovering from runtime issues, at least in a lab setting. This provides some reasoning to argue for the efficiency of the solution and to answer **RQ2** but as this workflow is created completely synthetically, as opposed to the other two workflows, for the purpose of this evaluation any argument towards efficiency in order to answer **RQ2** will be weaker for it compared to the others.

6.2 SARS-CoV-2 Genome Reconstruction

The following results stem from the execution of a real-world Nextflow workflow aimed to perform SARS-CoV-2 Genome Reconstruction on both vanilla Nextflow and the modified version. These results should give a better idea of how a solution like this could perform in the real world, with the caveat that this workflow had to be slightly modified to exhibit the specific type of bug covered by the retry strategy proposed in this document.

Average Dur.	Vanilla Nextflow	Modified Nextflow	Improvement
Overall	9.93m	8.04m	19.05%
No Errors	9.36m	7.91m	15.51%
One Error	10.43m	8.1m	22.34%
Two Errors	11.61m	8.52m	26.6%

Figure 10: Table collecting the averages of 826 executions of the SARS-CoV-2 Genome Reconstruction workflow with 413 executions performed on vanilla Nextflow and repeating until success when encountering an error and 413 times on the modified Nextflow version using the novel error recovery strategy.

The results show two distinct differences between the execution on vanilla Nextflow and the execution on the modified version of Nextflow. Firstly, the modified version of Nextflow seems to generally have a significantly better performance, even when no error was encountered at all (see figure 10). At the time of writing, there exists no explanation for this difference in performance. The solution alters the behavior of the software in case an error is encountered, but makes no significant changes to the behavior up until the first error is encountered. There are some slight changes to the way errors are handled and caching is performed, which theoretically could impact the best-case performance, but appear unlikely to do so. Inspecting the output data of the two different executions showed, that the outputs are equivalent and thus, the modified version does not appear to perform any less work. Another explanation might lie within the setup of the experiment. The different configurations were each executed on different virtual machines in a data center, it is possible that the set of machines that executed the vanilla workflow happened to perform worse, despite being of the exact same specifications.

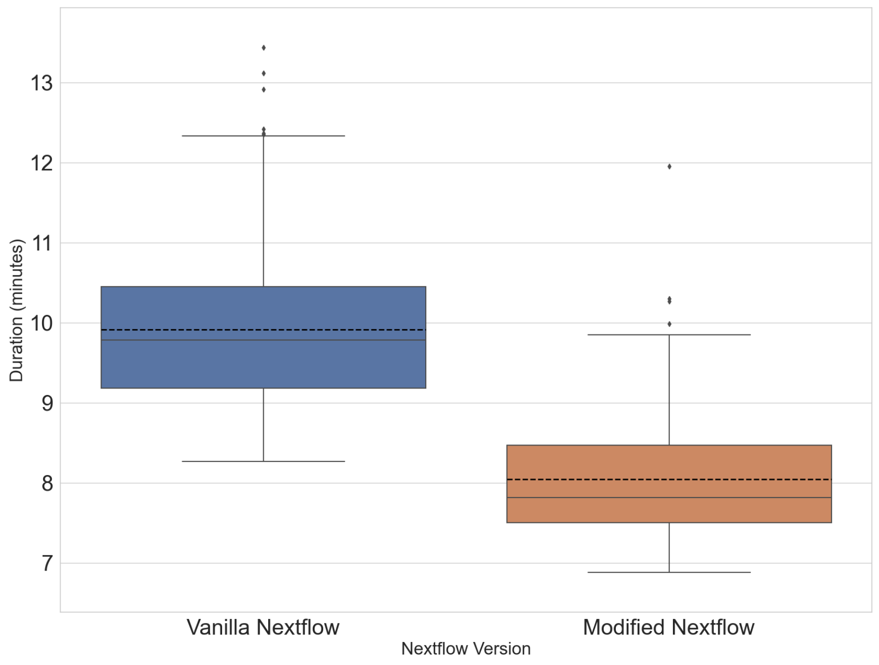

Figure 11: Boxplot collecting the results of 826 executions of the SARS-CoV-2 Genome Reconstruction workflow with 413 executions performed on vanilla Nextflow and repeating until success when encountering an error and 413 times on the modified Nextflow version using the novel error recovery strategy.

The second observation is that even when accounting for the shift in best-case performance, the modified version seems to perform better. The vanilla version of Nextflow has about a 4-minute span between the upper and lower whiskers (see figure 11) while the modified version only has about a 3-minute span between them. This difference is expected due to the behavior of the modified versions' error recovery strategy. Due to the caching, subsequent retries require less time than restarting the entire workflow and, thus, the worst-case performances with multiple errors encountered remain closer to the best-case performance.

Variance	Vanilla Nextflow	Modified Nextflow	Improvement
Overall	0.88m	0.53m	40.41%
No Errors	0.23m	0.43m	-87.3%
One Error	0.33m	0.53m	-59.1%
Two Errors	0.4m	0.64m	-60.07%

Figure 12: Table collecting the variance of 826 executions of the SARS-CoV-2 Genome Reconstruction workflow with 413 executions performed on vanilla Nextflow and repeating until success when encountering an error and 413 times on the modified Nextflow version using the novel error recovery strategy.

When looking at just the variance, it shows an overall improvement of about 40% due to the reduced cost of additional attempts compared to the baseline.

Overall the results show some improvement in efficiency over the baseline, even when not considering the general shift in performance. In regards to **RQ1** it is observable that the solution consistently recovers from the occurring runtime exceptions without any intervention and leads to successful completion while vanilla Nextflow is unable to do so for the same exceptions. In regards to **RQ2** an improvement in efficiency over the baseline is also visible, but the improvement is not nearly as significant as that of the synthetic video workflow. While this is a real-world workflow and thus the results produce a stronger argument towards the actual efficiency of the solution (**RQ2**), the results are not as optimal as they are for the synthetic video conversion workflow. The reason for this lies in the design of the workflow.

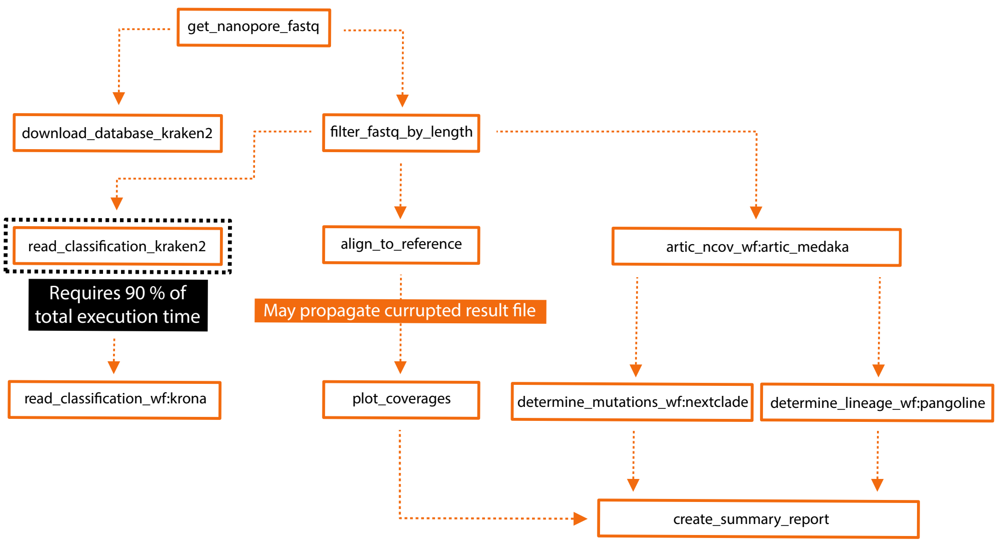
get_nanopore_fastq
download_database_kraken2
filter_fastq_by_length
read_classification_kraken2
Requires 90 % of total execution time
align_to_reference
artic_ncov_wf:artic_medaka
May propagate currupted result file
read_classification_wf:krona
plot_coverages
determine_mutations_wf:nextclade
determine_lineage_wf:pangoline
create_summary_report

Figure 13: One of the nodes in the workflow requires around 90% of the execution time of the entire workflow

One of the tasks in the workflow (highlighted in figure 13 takes up around 90% of the total execution time of the workflow, meaning if the workflow were to take 20 minutes to execute in total, 18 of those minutes would be spent on this task alone. If a grandchild of this task were to fail at runtime, then the parent retry strategy would retry one or more of the children of this long-running task but would not run this task again, saving 90% of execution time in subsequent runs. In this design of the workflow, this is, however, not possible as this task has no grandchildren. Because of this, the solution proposed by this document yields less impressive results for this workflow.

As this is a real-world workflow, the results are important and should be considered in evaluating whether or not such a solution is useful in a real-world context. In order to demonstrate however, that this scenario is somewhat arbitrary and might as well be different in the real world, a third workflow was created which formed a hybrid between a real-world and a synthetic one.

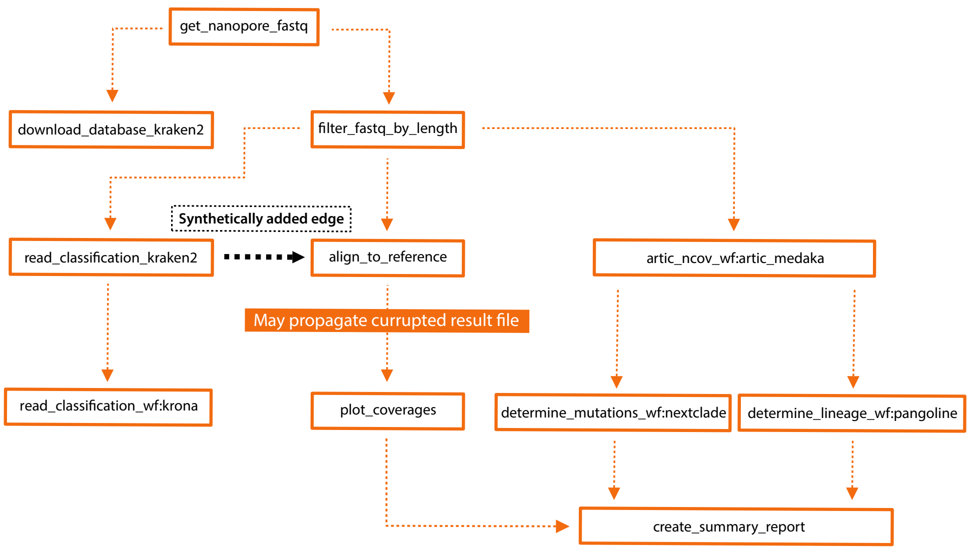
get_nanopore_fastq
download_database_kraken2
filter_fastq_by_length
Synthetically added edge
read_classification_kraken2
align_to_reference
artic_ncov_wf:artic_medaka
May propagate currupted result file
read_classification_wf:krona
plot_coverages
determine_mutations_wf:nextclade
determine_lineage_wf:pangoline
create_summary_report

Figure 14: Adding a synthetic additional edge in the workflow connection read_classification_kraken2 and align_to_reference.

The synthetic edge connecting read_classification_kraken2 turns the failing node plot_coverage into a grandchild node of read_classification_kraken2. With this new execution graph, when plot_coverage fails, its parent align_to_reference is scheduled for retry while read_classification_kraken2 is cached and, thus, taking full advantage of the parent retry strategy.

6.3 SARS-CoV-2 Genome Reconstruction with Artificial Edge

Running the solution again on the semi-synthetic new workflow, created by adding an artificial edge to the SARS-CoV-2 Genome Reconstruction workflow, demonstrates the solution's performance in a semi-lab setting.

Average Dur.	Vanilla Nextflow	Modified Nextflow	Improvement
Overall	13.72m	12.22m	10.99%
No Errors	10.55m	10.75m	-1.87%
One Error	16.59m	14.44m	13.01%
Two Errors	22.18m	14.07m	36.57%

Figure 15: Table collecting the averages of 812 executions of the SARS-CoV-2 Genome Reconstruction (Artificial Edge) workflow with 406 executions performed on vanilla Nextflow and repeating until success when encountering an error and 406 times on the modified Nextflow version using the novel error recovery strategy.

The results show that in this case, the modified version of Nextflow is indeed able to yield more favorable results in those cases where two errors are encountered but fails to do so for the other cases. This is unexpected and suggests an issue with the modified version of Nextflow produced by this work.

Variance	Vanilla Nextflow	Modified Nextflow	Improvement
Overall	20.67m	5.39m	73.91%
No Errors	1.71m	2.04m	-19.19%
One Error	4.54m	2.76m	39.28%
Two Errors	8.75m	2.5m	71.38%

Figure 16: Table collecting the variance of 812 executions of the SARS-CoV-2 Genome Reconstruction (Artificial Edge) workflow with 406 executions performed on vanilla Nextflow and repeating until success when encountering an error and 406 times on the modified Nextflow version using the novel error recovery strategy.

Even though variance is improved, it is not improved as significantly as in the other workflows (see figure 16). This also further points toward an issue with the implementation.

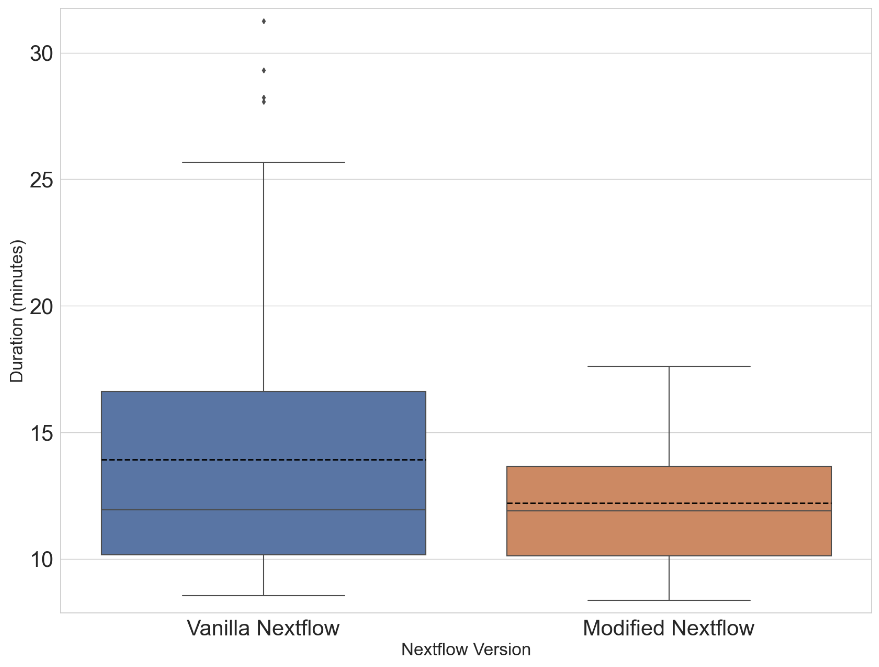

Figure 17: Boxplot collecting the results of 812 executions of the SARS-CoV-2 Genome Reconstruction (Artificial Edge) workflow with 406 executions performed on vanilla Nextflow and repeating until success when encountering an error and 406 times on the modified Nextflow version using the novel error recovery strategy.

The boxplot (see figure 17) shows there is a significant difference in the worst-case performances, but only a slight difference in the average duration and no difference in the median duration.

The average durations by number of errors 15 uncover some unexpected results. There is a significant jump of about 4 minutes between those executions that succeeded on the first attempt and those that succeeded on the second attempt. This is unexpected and not consistent with the results for the other workflows. Most of the work done in the first attempt should be transferred to the second attempt, leaving only some operational overhead as additional duration. In this case, the overhead of 4 minutes, which, given the duration of the best-case, amounts to 40% of the duration, is significant and unexpected. As Nextflow is a highly complex software dealing with scheduling, concurrency, and third-party dependencies such as docker, finding the cause for those results is not trivial. In the scope of this work, it was not possible to explain these results. It is possible that this workflow leads to an edge case where the injection of the caches leads to some kind of deadlock or delayed scheduling in Nextflow, which causes the cached runs to have such significant overhead. Despite these anomalies, however, the results demonstrate once again an improved performance of modified Nextflow over the baseline.

The effectiveness of the solution remains consistent for this workflow and thus, the issues uncovered by these results remain consistent with what has been observed with the other workflows in regards to **RQ1**. As the results still indicate that the solution is more efficient than the baseline, the results also remain consistent with previous results in regards to **RQ2**. The unexpected behavior and noticeable drop in efficiency, compared to the other workflows, does however point towards a more conservative answer to **RQ2**, where the solution can be highly efficient in some cases, but more moderately efficient in others.

6.4 Bacterial Genomes Workflow

This workflow from the field of bioinformatics is designed to sequence bacterial genomes. Next to the SARS-CoV-2 Genome Reconstruction workflow, it functions as the second real-world example with the caveat that a propagated bug was introduced on purpose.

Average Dur.	**Vanilla Nextflow**	**Modified Nextflow**	**Improvement**
Overall	5.8m	4.08m	29.55%
No Errors	3.91m	4.04m	-3.38%
One Error	7.27m	4.13m	43.17%
Two Errors	10.27m	4.19m	59.16%

Figure 18: Table collecting the averages of 666 executions of the Bacterial Genomes workflow with 333 executions performed on vanilla Nextflow and repeating until success when encountering an error and 333 times on the modified Nextflow version using the novel error recovery strategy.

The results show a significant difference in performance for all error cases, due to the fact that the caching of the modified Nextflow reduces the cost of the rerun to a fraction of the full run (down to less than 30 seconds from about 3m), whereas the baseline requires a full rerun and increases durations in about 3m steps depending on the number of errors that have occurred. This leads to an overall improvement for the average duration of 29.55% and as (see figure: 18) shows, this improvement is increasingly higher, the more errors are encountered.

Variance	Vanilla Nextflow	Modified Nextflow	Improvement
Overall	5.7m	0.02m	99.68%
No Errors	0.12m	0.01m	87.87%
One Error	0.46m	0.02m	96.01%
Two Errors	0.57m	0.01m	98.16%

Figure 19: Table collecting the variance of 666 executions of the Bacterial Genomes workflow with 333 executions performed on vanilla Nextflow and repeating until success when encountering an error and 333 times on the modified Nextflow version using the novel error recovery strategy.

The variances of samples in (see figure 19) show, that there is significantly less variance for the modified version of Nextflow. As the table shows, the variance is largely due to the difference in execution time when different amounts of errors are encountered. The variance remains low in both versions when looking comparing samples only with other samples with the same error count.

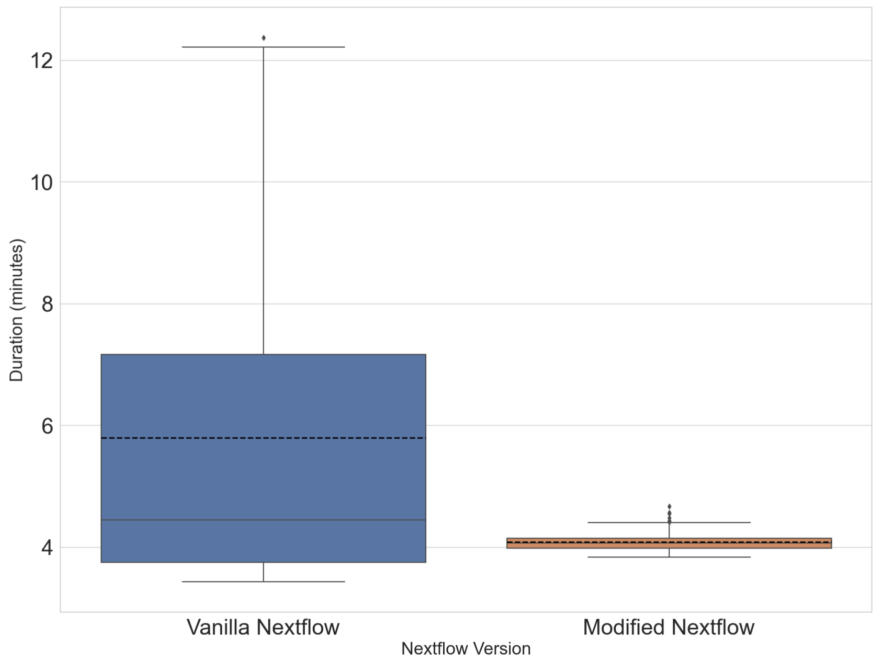

Figure 20: Boxplot collecting the results of 666 executions of the Bacterial Genomes workflow with 333 executions performed on vanilla Nextflow and repeating until success when encountering an error and 333 times on the modified Nextflow version using the novel error recovery strategy.

The boxplot (see figure 20) further illustrates this. The "Bacterial Genomes Workflow" has significantly less variance in the recorded execution durations compared to the baseline, making the execution time appear almost constant next to it, even with multiple retries. The mean of the baseline is significantly higher than the mean of the modified version. The execution times further diverge when comparing runs with more errors and subsequent retries, as evidenced by the large distance between the whiskers where the baseline worst cases reside around 12 minutes and the modified version worst cases still reside under 5 minutes, making it more than twice as efficient.

Again the solution demonstrated to be effective in circumventing bugs that were propagated from parents nodes while vanilla Nextflow wasn't showing that the solution was consistently effective in circumventing propagated bugs throughout the entire evaluation (**RQ1**). With an almost 30% improvement in average runtime, compared to the baseline, the solution proves to yield a significant improvement in efficiency for this particular workflow. As this is a real-world workflow the significant performance improvement observed helps strengthen the argument to answer **RQ2** with a yes.

6.5 Evaluation Summary

Workflow	Any Errors	No Errors	One Error	Two Errors
Video	24.11%	1.18%	43.14%	48.84%
Covid	19.05%	15.51%	22.34%	26.6%
Covid (Art. Edge)	10.99%	-1.87%	13.01%	36.57%
Genome	29.55%	-3.38%	43.17%	59.16%
Overall	20.92%	2.86%	30.42%	42.79%

Figure 21: Table collecting and aggregating the average improvements across all four workflows that were part of the evaluation.

Despite the issues that surfaced in some of the evaluations, the solution still yields a significant overall improvement of 20.92% for the average performance. When considering only those cases where at least one error occurred, the improvement is as high as 36.6%.

Workflow	Any Errors	No Errors	One Error	Two Errors
Video	96.82%	64.22%	88.04%	-
Covid	40.41%	-87.3%	-59.1%	-60.07%
Covid (Art. Edge)	73.91%	-19.19%	39.28%	71.38%
Genome	99.68%	87.87%	96.01%	98.16%
Overall	77.7%	11.4%	41.06%	36.49%

Figure 22: Table collecting and aggregating the variance improvements across all four workflows that were part of the evaluation.

It is also evident that variance overall is reduced by an average of 77.7%, which demonstrates that the cost of re-execution is greatly reduced, making the overall durations more consistent even with error occurrences.

6.5.1 Reasearch Questions

Based on the evaluation results the research questions can be revisited:

RQ1. Is it possible and feasible to implement an error recovery strategy that restarts parent nodes of repeatedly failing nodes in the Nextflow code base?

RQ2. Is the restarting of parent nodes of repeatedly failing nodes a viable strategy to circumvent bugs that are propagated from parent nodes?

Answer to RQ1 For all three workflows the modified version of Nextflow was effective in successfully completing execution despite the occurrence of propagated bugs that led to runtime exceptions (6.1, 6.2, 6.4). This validates the modified version of Nextflow proposed by this work as a proof of concept, thus demonstrating the possibility of such a solution. Given the time constraints of this bachelor's thesis, it is also demonstrated that building such a solution is feasible in any reasonable scope.

Answer to RQ2 As, so far, there are no existing error recovery strategies that are able to recover from bugs that were propagated from a parent node 3.3 the only existing solution is to restart the entire workflow from scratch. While the evaluation uncovered unexpected and unexplained behavior (6.3) and some results were more impressive than others (6.4 vs 6.2), the evaluation in this work was able to demonstrate that the novel error recovery strategy of restarting parent nodes was consistently significantly more efficient than the existing solution (6.1, 6.2, 6.4). For the workflows tested, the novel error recovery strategy applied in a modified version of Nextflow proved to be more efficient than the baseline and thus a viable strategy to recover from the targeted class of errors.

7 Threats to Validity

As mentioned in previous chapters (5.1) the evaluation is limited and thus is vulnerable to external threats to validity. Additionally, the evaluation produced some unexpected results, which point towards either issues in the software or in the experiment posing internal threats to validity.

7.0.1 Limited Number of Workflows in the Evaluation

Due to several factors such as the limited availability of (functioning) public workflows or time constraints of this work, the evaluation was performed on only a set of three different samples. It is highly unlikely that this sample size accurately represents the population of real-world nextflow workflows and thus, the small number of the test dataset poses a threat to external validity.

7.0.2 Synthetic Nature of the Workflows

In two of the workflows a propagated bug was introduced intentionally and the third workflow was entirely created synthetically, including a propagated bug. Additionally, the bugs were purposely made to occur in 50% of all cases. This synthetic nature of the workflows poses a threat to external validity as the ability to generalize to real-world bugs and real-world workflow behavior is compromised.

7.0.3 Potential Issues in the Experiment Setup

Some of the executions (see 6.2 and 6.3) produced results that were not entirely explainable. The reason for the unexpected nature of the results could lie in the experiment setup and may include unexpected differences in the hardware provided by the cloud host, differences in the speed of the internet connection of these machines, or any other unexpected environmental difference in the setup. The unexpected nature of some of the results potentially points to a threat to internal validity as the setup may have been different for different configurations.

8 Conclusion

This work attempted to demonstrate the possibility and viability of a novel error recovery strategy that recovers from propagated bugs by restarting parent nodes of failing nodes. While there are some limitations to the research, nevertheless, the work was able to demonstrate two things. First, if a propagated bug that is both fixable through retrying and is propagated from a parent node, exists in a workflow, then the solution is effective at recovering from that bug. Secondly, for the set of workflows on which the solution was evaluated on, the solution proved to yield an improvement in efficiency over the baseline. While the bugs were introduced precisely and intentionally, Murphy's law tells us that, if a type of error is possible, which this research demonstrates, it will eventually occur. Whether these types of bugs occur often enough to warrant the potential overhead of a novel automatic error recovery strategy, and whether the efficiency of the solution observed in this work translates to the real world remains to be seen. What has been shown, however, is that there is promise and that such a solution may be worth investing in further.

9 Future Work

A valuable next step for future research would be to evaluate the error recovery strategy in real-world scenarios through ongoing use or by employing a large set of Nextflow workflows without selection bias or modification. If future research were indeed able to demonstrate, that parent retrying is a viable error recovery strategy in the real world of scientific workflow management then further extensions to the solution could be evaluated:

1. Is there a heuristic that would allow for a decision on whether or not to apply the solution at runtime, to avoid unnecessary overhead?
2. What if a bug was propagated through multiple nodes? Is restarting grandparents a viable strategy?
3. What is the underlying issue that is causing the unexpected results of the workflow "SARS-CoV-2 Genome Reconstruction with Artificial Edge"? How would the solution perform if this issue were resolved?

Bibliography

[1] Felix C. Gärtner. "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments". In: *ACM Computing Surveys Volume 31, Issue 1* (1999). DOI: 10.1145/311531.311532.

[2] Mouallem and Pierre A. "A Fault Tolerance Framework for Kepler-based Distributed Scientific Workflows". In: *B. (eds) Scientific and Statistical Database Management. SSDBM 2010. Lecture Notes in Computer Science, vol 6187* (2011). DOI: 10.1007/978-3-642-13818-8_31.

[3] Carl Kesselman Soonwook Hwang. "Grid Workflow: A Flexible Failure Handling Framework for the Grid". In: *Conference: 12th International Symposium on High-Performance Distributed Computing (HPDC-12 2003)* (2003). DOI: 10.1109/HPDC.2003.1210023.

[4] Jia Yu and Rajkumar Buyya. "A Taxonomy of Scientific Workflow Systems for Grid Computing". In: *SIGMOD Rec.* (2005). DOI: 10.1145/1084805.1084814.

[5] Sajjad Haider and Babar Nazir. "Fault tolerance in computational grids: perspectives, challenges, and issues". In: *SpringerPlus 5* (2016). DOI: 10.1186/s40064-016-3669-0.

[6] Brian Randell. "System structure for software fault tolerance". In: *SIGMOD Rec.* (2005). DOI: 10.1145/1084805.1084814.

[7] Adam Beguelin, Erik Seligman, and Peter Stephan. "Application Level Fault Tolerance in Heterogeneous Networks of Workstations". In: *Journal of Parallel and Distributed Computing* (1997). DOI: 10.1006/jpdc.1997.1338.

[8] Ewa Deelmana et al. "Pegasus: A framework for mapping complex scientific workflows onto distributed systems". In: *Scientific Programming* (2005). DOI: 10.1155/2005/128026.

[9] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: NSDI'12 (2012).

[10] Juan Leon, Allan L. Fisher, and Peter Steenkiste. "Fail-safe PVM: A portable package for distributed programming with transparent recovery". In: *IEEE Symposium on High Performance Distributed Computing* (1994). DOI: 10.21236/ada266594.

[11] Gopi Kandaswamy, Anirban Mandal, and Daniel A. Reed. "Fault Tolerance and Recovery of Scientific Workflows on Computational Grids". In: *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Lyon, France* (2008). DOI: 10.1109/CCGRID.2008.79.

Appendix

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird. Berlin, den

August 4, 2023 ....