MASTER'S THESIS 2025

Early Defect Detection Through Ephemeral Preview Environments

Björn Tenje Persson, Fredrik Wastring

LU-CS-EX: 2026 - 11

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE

Datavetenskap

LU-CS-EX: 2026 - 11

Early Defect Detection Through Ephemeral Preview Environments

Tidig Defektdetektering Med Temporära Förhandsvisningsmiljöer

Björn Tenje Persson, Fredrik Wastring

Early Defect Detection Through Ephemeral Preview Environments

(A mixed-methods embedded case study)

Björn Tenje Persson
bjorn@tenje.se

Fredrik Wastring
fredrik@wastring.com

April 24, 2026

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors:
Ulf Asklund, ulf.asklund@cs.lth.se
Sebastian Nordin, sebastian.nordin@ifacts.se
Malin Jeppsson, malin.jeppsson@ifacts.se

Examiner:
Sule Tekkesinoglu, sule.tekkesinoglu@cs.lth.se

Abstract

Ephemeral preview environments (EPEs) provide production-like validation for trunk-based development workflows, but are seldom evaluated in highly configurable, customer-specific software. This thesis presents a case study at iFACTS AB, where limitations in local development setups and shared staging environments were associated with defects escaping into production. To address this, we designed and implemented a preview-environment platform consisting of a web portal, GitOps-driven environment orchestration using Argo CD, Kubernetes, Helm, and a set of curated customer dataset templates. We also propose a new workflow that inserts a pre-merge "preview gate" that forces branch-specific deployments with representative data to be validated before being merged to the trunk. The solution was evaluated using an embedded, mixed-method approach using Technology Acceptance Model constructs to gauge adoption, together with PERT-based estimates to assess avoided work. The participating developers and QAs reported high confidence in the tool regarding usage and setup, with concerns about dataset freshness and minor UI problems. Across three micro-demo cases, the evaluation suggested an average avoided rework of 5.5 hours per story. While the study is limited by a small sample size and a single-organization context, the results indicate that EPEs can move defect detection earlier in the process when supported by solid automation and data ownership.

Keywords: Software testing, Ephemeral preview environments, Customer-specific validation, Kubernetes, GitOps, Technology Acceptance Model, Pre-merge testing

Contents

Chapter 1
Introduction

This thesis was conducted at iFACTS AB, a company striving to improve its software development processes through initiatives across multiple fronts. One area under active change is how the company handles source code, particularly branching strategies. Currently, they use a workflow similar to Microsoft's Release Flow combined with GitHub flow: developers work in branches that are merged into the main branch, and production releases are cut from a release branch off main [1], [2], [3]. The long-term direction is trunk-based development, where integration happens continuously and changes are kept small, with a goal of eventually moving toward direct-to-trunk commits. Keeping the trunk (main) branch continuously deployable supports shorter feedback cycles and more frequent releases [4], [5].

Three drivers shape the problem framing in this thesis: setup cost, environment fidelity, and environment isolation. First, environment replication is costly. Manual setup requires technical expertise and domain knowledge to enable and configure customer-specific features for validation. Second, local development environments have limited fidelity. They validate features in isolation but miss defects that only emerge in production-like conditions, such as configuration mismatches, data-dependent edge cases, and integration issues with external services. Third, shared staging environments are not isolated. Parallel QA activity can introduce data drift and interference between testers.

These drivers combine into a causal chain: the high setup cost discourages micro-demos and pre-merge validation; skipped validation allows defects to escape into the trunk branch; and those defects are later discovered in shared staging or production, where they are more expensive to resolve.

To address these issues, this thesis focuses on *ephemeral preview environments* (EPEs). EPEs are temporary, branch-scoped environments created from templates and representative datasets. They allow stakeholders to validate features with production-like configuration and data without interfering with other environments. The goal is to use EPEs as a supplement to code review and micro-demos to reduce defects that would otherwise reach staging or production.

To prevent defects from escaping the branch, a repeatable preview environment must be tied to a single code branch and a customer-relevant dataset. This enables collaborative validation

before merging and supports early defect detection that would otherwise be delayed until later stages. EPEs provide this by provisioning short-lived, branch-specific environments on demand without risking shared staging data.

1.1 Research Goals

To be able to pinpoint what the case company needs for a solution, we must first figure out what the limitations of the current process are. After understanding the limitations of the current processes, we investigate what solutions and workflow changes can be made to the current process. Finally, we evaluate the solution together with the new process to see if they achieve any tangible improvements in the amount of rework time that is saved by catching bugs earlier. This leads us to the following research questions:

- **RQ1**: What are the limitations of the current process for provisioning and configuring internal development and test environments?
- **RQ2**: What technical solutions and workflow modifications can be implemented to address the issues addressed in RQ1?
- **RQ3**: How useful and easy-to-use do the developers perceive the proposed workflow to be?
- **RQ4**: Within the iFACTS case study context, how much rework time is avoided when customer-specific defects are detected in EPEs before merge, compared with later detection after merge to trunk?

1.2 Report Structure

Chapter 1 provides a background into the thesis topic, introduces the problem, and gives a brief overview of the thesis's structure. Followed by Chapter 2, which presents relevant theoretical background to ensure the reader has the necessary knowledge to understand the thesis. The implementation process is explained in Chapter 4, giving an insight into architectural decisions, code-level considerations, and encountered challenges. The solution is then evaluated, validated, and compared to related works in Chapter 5. We then discuss the validity of the evaluation together with discussions of methodological choices and the implications of the findings in Chapter 6. Finally, Chapter 7 outlines the conclusion as well as possible future work.

Chapter 2

Background

To understand the contents of this thesis, some knowledge of iFACTS' development and testing workflow and hosting environments is required. After this, we delve deeper into specific tools that are used in the context of EPEs. Finally, we situate these concepts in the context of related works that have explored similar solutions.

2.1 Software Delivery and Environment Management

2.1.1 CI/CD

Continuous Integration (CI) is the practice of integrating changes to the codebase automatically and as often as possible, for example, whenever a feature or bug-fix is complete [5].

The way iFACTS uses CI differs, but they use it mainly to run automatic unit and integration tests on commits to the main branch. The CI pipeline also builds a *container image* on each commit to the branch. A container image is a static file that contains all the needed components to run a *container*, which is an executable unit of software.

After the container image is built, it is pushed to a *container registry* where all the images are stored along with the hash of the commit it was built from.

One thing that is missing from the CI pipeline is the testing of customer-specific variants of the software before merging to the trunk.

2.1.2 Configuration Drift

Configuration drift refers to a gradual change of an environment from its intended state due to cumulative changes over time. The phenomenon is especially common in shared, long-lived environments [6].

In our case, the databases for the staging environments were frequently changed by multiple testers conducting parallel regression activities. This results in a database state that is difficult to manage, due to the uncoordinated nature of many concurrent changes.

2.1.3 GitOps

GitOps is an operating model where the desired state of infrastructure and applications is declared in version-controlled files, and an automated reconciler continuously applies that state to the runtime system [7].

In this thesis context, GitOps is relevant because it supports repeatable provisioning, traceable state changes through commit history, and automatic drift correction. These properties are central when creating short-lived preview environments from branch-specific configuration.

2.1.4 Provisioning vs. Configuration

The operations team at iFACTS has recently started using Kubernetes, which is an open-source system for automating the deployment and management of containerized applications [8]. Kubernetes runs on multiple computers together, on what is known as a cluster.

At iFACTS, the EPEs are built with a software called Argo CD, which periodically polls a Git repository and looks for changes in the state of the configuration files. If it finds any changes to the monitored files, it will update the state of the cluster to reflect these changes.

By using this approach in managing the internal environments, configuration drift is reduced, since the desired state is declared explicitly in the files, as opposed to imperatively in the software itself. This also enables automated setup of environments, since the configuration files are easy to copy and modify for new environments.

2.2 iFACTS Development Workflow

We will start off by explaining parts of the iFACTS workflow, which is important to understand the thesis.

2.2.1 Stories

At iFACTS, the development workflow revolves around stories. If something needs to be implemented, i.e., a feature or an issue has been detected, a story will be created by a stakeholder. A ticket will be created in Jira and will then be assigned to a developer. This story will contain all of the necessary information for the developer to implement the story and for a tester to

perform all of the necessary testing. During development, a separate branch will be created by the developer, which is merged when the story is marked as complete.

2.2.2 Micro Demos

Before a story can be marked as complete and merged into the trunk branch, the developer is required to perform a *micro demo*. This demonstration is semi-standardized and takes place on Google Meet or Slack Huddles. It includes a Business Analyst (BA), Quality Assurance (QA), and other relevant stakeholders. The purpose of the demonstrations is for the developer to demonstrate that the story has been fully implemented, identify potential bugs or issues, and gather feedback on potential improvements.

To perform the micro demo, the developer needs to have a fully working and configured environment set up, which includes the code they have written. At this point, the only way for the developers to do this is to use their own development environments, in which they have already completely tested the code.

2.3 Ephemeral Preview Environments

Ephemeral Preview Environments (EPEs) are used in the industry as short-lived environments that are deployed automatically or on demand [9]. They do not have to be manually configured, neither in infrastructure nor in data. Their main characteristic is that they are destroyed when their purpose has been served.

2.4 Evaluation and Adoption

The Technology Acceptance Model (TAM) is a theoretical model that is used to predict the user acceptance of technology [10]. TAM uses two primary factors to determine whether or not individuals will adopt a new technology. The Perceived Usefulness (PU), which refers to how much a user believes a technology will improve their job performance, and Perceived Ease of Use (PEOU) refers to the amount of effort required to interact with the technology. The combination of the factors creates the behavioral intention to use the product.

2.5 Related Works

Much of the related work that directly relates to EPEs comes from industry articles in the context of hosting costs and simplifying the delivery of custom environments, such as online articles and blog posts that summarize how they are used in different companies. However, because these topics have a strong connection to DevOps and CI/CD, there is substantial information to draw from research in those fields.

2.5.1 EPEs vs. Staging Environments

The core mechanism that is being studied in this thesis is the differences in usage of EPEs and traditional staging environments. This mechanism is key for detecting faults in the software earlier in the software delivery process, since the faults can be caught before being merged into the trunk as opposed to after. The goal of the modern EPEs is software isolation, reduced configuration drift, and faster feedback on code changes. All of these advantages are repeated in many industry cases such as Netlify, GitLab, and Vercel [9], [11], [12], [13].

Other academic articles also align with the advantages that these articles propose. Shrestha et al. argue that configuration drift leads to operational inefficiencies and security vulnerabilities [7]. Qu et al. propose that performance and behavior can shift across different configurations in highly configurable systems [14]. This is also strengthened by both Hilton et al. and Vasilescu et al., who show the benefits of early fault detection using CI practices [15], [16].

However, there does not exist any empirical comparison between the preview environment and staging environments for early defect detection in highly configurable software. We intend to address this by estimating the amount of time that is saved by resolving the defect in the branch, as opposed to detecting and resolving the defects after the feature has been merged to the trunk.

2.5.2 Testing Highly Configurable Systems

The task of testing highly configurable systems is complicated because the option space explodes with each possible option. In our use case, this spans both system-level variants per customer as well as user-level feature toggles.

Combinatorial interaction testing (CIT) has been used extensively in research about configurable systems to find optimal sets of options when testing configurable software [17], [18]. Studies on configuration-aware regression testing show that sampling and prioritizing certain configurations can improve early fault detection compared to random sampling [14]. These works focus on automated test suites that are run to ensure that the software quality is high.

However, most prior work handles the code-level option space while our case spans system configurations with user-level configuration added to it [14], [19], [20]. They also differ from our use case in that they focus on automated test suites to try to test all variants using covering arrays, while our approach acts as a supplement to a specific kind of pre-merge testing where we want to test a customer-specific variant.

Our work addresses this gap by providing new research on how a selected customer-specific configuration that combines system-level and user-level options can be exercised in *ephemeral preview environments* as part of pre-merge testing to reduce escaped defects.

2.5.3 Impact of CI Practices

Continuous integration (CI) increases the speed of feedback and, in turn, defect discovery. This matters for us because previews will only add value if they are exercised quickly after each change. Our case company already uses forms of CI for automated tests, unit tests, and integration tests. But they do not include this kind of variant testing in their CI, and they only test one variant.

Empirical studies have found that CI adoption correlates with generally improved software quality [15], [16], [21], [22]. The focus differs on what kind of CI adoption is implemented, that is, whether it is CI sub-practices, such as build quality, or broader CI measures.

The way that our thesis differs is that we measure how integrating preview environments into the CI pipeline can change the general defect lead time and the avoided rework time. This differs from traditional CI research in that it focuses more on how preview environments can be used to improve CI practices instead of how CI practices improve software quality.

Chapter 3

Limitations of the Current Process

> What are the limitations of the current process for provisioning and configuring internal development and test environments?
>
> — RQ1

This section addresses RQ1 by identifying and structuring the current limitations of the process for provisioning and configuring internal environments for development and testing. Our findings are derived from informal interviews with key stakeholders and employees at the case company, and are organized into categories based on the effect they have on developer and QA workflows.

We talked to and interviewed employees from different positions, such as the Chief Technical Officer (CTO), the Operations Manager (OM), the Business Analyst (BA), some testers, and some developers, to fully understand the limitations of the current process.

3.1 Fragmented and Costly Environment Replication

A recurring limitation that was brought up by both management and developers was the high cost and effort of replicating the development environment for validation purposes. Micro-demos, which were intended as lightweight validation checkpoints, were often avoided due to the effort required to replicate the development environment needed to represent the feature. Many developers perceived this effort as disproportionate to the scope of the features, leading to a reduced interest in participation.

3.2 Delayed Detection of Integration Defects

Another recurring theme that was noticed is the delayed detection of defects that only appear in the shared staging environments. One of the main reasons for this is that developers primarily validate their changes only in isolated local setups that might have discrepancies in configuration and data.

This delay increases the cost of defect resolution by forcing the developers to revisit the code after they have already merged it and moved on to something else. They then have to recreate their developer environment with the correct configuration of data, which also takes time.

3.3 Shared Environment Contention and Data Fragility

Members of the QA team brought up challenges regarding testing in the shared staging environments. Since these environments contain data that is mutable and accessed concurrently by different members of the testing team, testers must always consider the risk of corrupting the data and thereby interfering with other test activities.

These constraints limit the scope of the testing of features and could introduce hesitation when executing more destructive or exploratory test cases. This limits the testers from fully testing the feature, which could lead to defects escaping into the production environments, which would be even more costly.

3.4 Summary of Identified Limitations

Taken together, these recurring limitations reveal a development and testing process that lacks the flexibility, isolation, and timely feedback that is needed. Developers are limited in their ability to validate their changes in a production-like context without having to merge the code to the trunk early. The QA team is limited by shared staging environments that are fragile and coupled to a specific branch structure.

These issues together motivate the need for a way to create environments on demand, where the environments can be instantiated with a specific code version and a data set that is relevant and representative of a customer environment. This approach enables early, production-like validation for the developers and isolated and reproducible testing for the testers. These requirements informed many of the design goals for the implementation phase, which is detailed in Chapter 4.

3.5 Derived Requirements for the Solution

Based on the identified limitations, we derived the following requirements that the proposed solution should satisfy:

- **R1 - Fast, repeatable provisioning:** A preview environment should be created from a branch and a selected dataset without manual infrastructure setup.
- **R2 - Production-like validation context:** The environment should support customer-relevant configuration and representative data for the targeted variant.
- **R3 - Environment isolation:** Validation and QA activities should not interfere with each other through shared mutable state.
- **R4 - Lifecycle control and auditability:** Environments should be straightforward to update, reset, and delete, and state changes should remain traceable.

Chapter 4

Solution

> What technical solutions and workflow modifications can be implemented to address the issues addressed in RQ1?
>
> — RQ2

The chapter follows the journey of an EPE from the moment it is requested to the point where it is a live environment. We first map the system architecture to understand each moving part. We then walk through the user-facing portal, the orchestration layer, and the dataset templates that will turn input into infrastructure. With that foundation in place, we discuss the implementation decisions that were made and finally conclude with the EPE-enabled workflow recommended for teams.

4.1 Design Requirements, Non-goals, and Traceability

To keep the design rationale explicit, we map the requirements from Chapter 3 to concrete solution decisions.

Requirement	Motivation from RQ1	Design response in this thesis
R1	Environment replication is costly and discourages pre-merge validation.	Portal-driven request flow + GitOps provisioning through Argo CD and Helm templates.
R2	Local setups miss data- and configuration-dependent defects.	Mandatory customer dataset template selection and branch-specific deployment.
R3	Shared staging creates contention and fragile mutable state.	Short-lived, branch-scoped environments with independent lifecycle and reset support.
R4	Validation workflows need predictable operations and accountability.	Git as source of truth, JSON-based state, TTL cleanup, update/reset/destroy actions with commit history.

Table 1: Traceability from identified requirements to implemented design choices

The following non-goals delimit the scope of the implementation:
- Full automation for creating an environment on every pull request.
- Organization-wide rollout and long-term field evaluation during the thesis timeframe.
- Building a custom Kubernetes Operator/CRD platform for environment orchestration.

4.2 System Architecture Overview

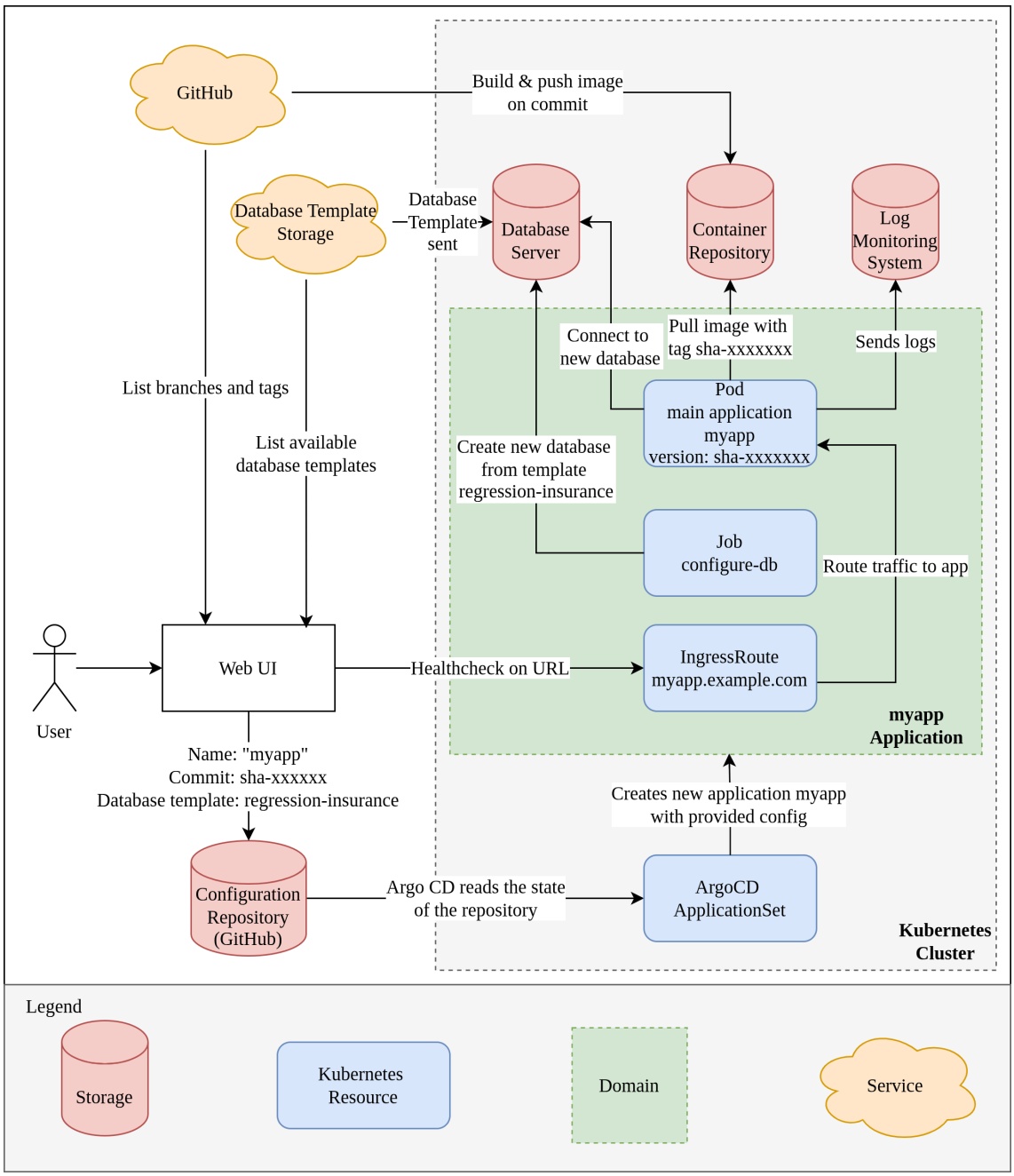

Figure 1: An overview of the system architecture of the Preview Environment Platform

Figure 1 shows how these components interact in practice. The system works such that an action in the portal writes to a configuration file in the repository. After this, Argo CD notices the change and uses Helm to generate the Kubernetes manifests needed. In turn, the cluster schedules the resulting workloads on the appropriate nodes. This flow is the backbone of the subsequent solution that the rest of the chapter unpacks.

4.3 Preview Environment Portal

To create the EPEs, the users need an interface. We chose to develop a custom Web UI due to the ease of access, as well as existing user familiarity with these kinds of interfaces. This also enabled us to use in-house authentication, as we could implement it on an already existing internal webpage that used central authentication.

Other alternatives to creating a custom Web UI were either using a ready-built solution such as Backstage, which is an open source framework for building developer portals, or connecting it with GitHub Deployments using GitHub Actions for automation [23], [24]. However, both of these solutions would have required substantial configuration to support our workflow. Given the limited scope and time frame of the project, we prioritized a focused proof-of-concept. Building the Web UI ourselves allowed us to control the complexity and only implement the features needed to evaluate the new EPE workflow.

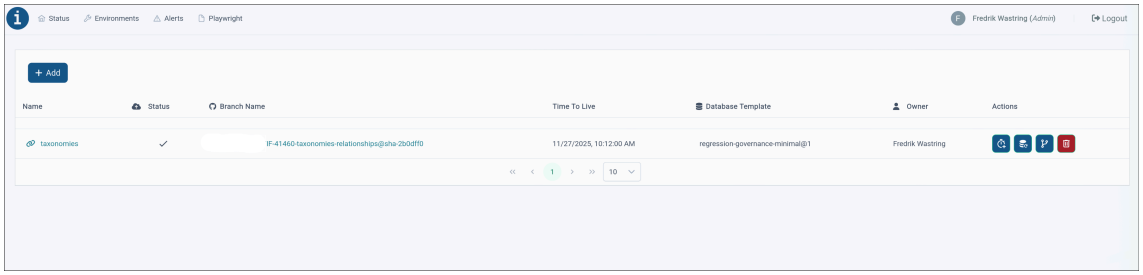

Figure 2: The frontend of the environment portal with one environment

The portal is an interactive webpage, as can be seen in Figure 2, that the API keeps in sync with all active EPEs from a GitHub repository. Each row of the table represents an individual environment. The columns include the name of the environment and the status of deployment. As well as which branch and commit the source code is based on, which database template the environment uses, how long the environment has before it is removed, and finally, a set of buttons with actions for lifecycle management.

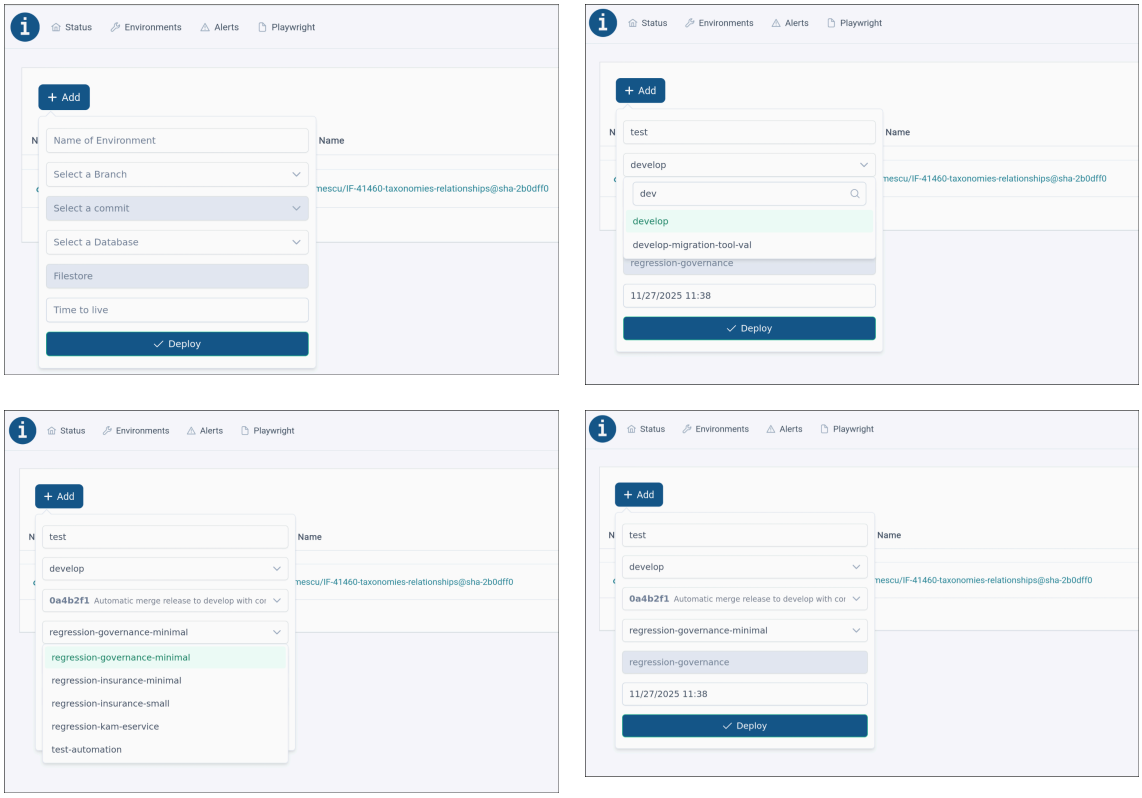

Figure 3: The form for creating a new environment with the branch and database drop-downs

To display the list of active EPEs, the API requests all of the JSON files from the previously mentioned repository. The data in the JSON files is then used to populate the table, with the exception of fields that are calculated at runtime, such as the status of the environment. The status of the environment is calculated by trying to access the environment using the JavaScript library fetch. If a 200 response is received, then it is deemed to be active, and otherwise it is seen as loading. This check is done on load and after that every 30 seconds, so as not to overload the environments. Other approaches for this are either using Server-Sent Events (SSE), where the server updates the view after receiving the information, or polling, but using incremental back-off. We chose normal polling due to the engineering complexity of SSE, where it would have caused a lot of overhead.

Another alternative to using the JSON file in a Git repository is to use API-driven state or to use custom Kubernetes CRDs. The reason that we chose not to use these alternatives is due to the great integration that Argo CD has with Git files structured as JSON for Application generators [25]. If it were not for the fact that the case company already had experience using Argo CD Applications, then using custom CRDs would probably be a wiser choice [26]. However, the overhead of developing custom CRDs would also be quite large, so the scope would not fit in with the thesis.

When a user wants to create a new environment, the user clicks the “New” button and proceeds to fill in the name, the code branch, the specific commit, the database template, and the “Time To Live” for the environment. The API requests the branch and commit data directly from GitHub’s API, while the list of database templates is requested from an internal file storage service. When the user fills out the form correctly and deploys, the API commits and pushes a

JSON file containing the aforementioned information to a specific GitHub repository. The whole flow can be seen in Figure 3.

```
{
  "environment": {
    "branch": "developer/IF-12345-generic-oidc",
    "name": "test-environment",
    "hash": "sha-c936584"
  },
  "templates": {
    "database": "regression-insurance-minimal",
    "filestore": "regression-insurance",
    "db_rev": 1
  },
  "metadata": {
    "owner": "Fredrik Wastring",
    "ttl": "...",
    "created": "..."
  }
}
```

Listing 1: An example of a config.json environment configuration

Listing 1 illustrates an example of a JSON configuration file. The fields for environment and template are going to be used by the orchestration layer to create the environment, while the metadata field is used by the portal to display ownership of the environment, as well as the "Time To Live" (TTL) calculations. The TTL is calculated by the API, and if the current date is later than the specified TTL of the environment, then the API deletes that JSON file from the repository, which the ApplicationSet sees and deletes the Application.

Each row of the table has action buttons in which a user can extend the lifetime of the environment, reset the database to a clean slate, change the commit that the code is based on, or immediately destroy the environment. When a lifecycle management button is clicked, a request is sent to the GitHub API that patches the file that the environment is represented by. For the destroy button, the API deletes the file. For the restore-database button, the db_rev field gets incremented, which updates a corresponding field in the Application, which in turn triggers a Kubernetes Job that restores the database to a clean slate. Finally, for the change commit button, the API updates the environment.hash field to the latest commit hash of the branch, which is propagated to the Application, and which then changes the Kubernetes Deployment to the new image tag.

After a configuration file is committed to the GitHub repository, the orchestration layer takes control and translates it to Kubernetes resources in the next section.

4.4 Environment Orchestration

Many essential parameters, such as networking and storage, will have the same configuration for each environment. Therefore, our setup will benefit greatly from the declarative and repro-

ducible nature of Kubernetes. It will allow us to create each environment with the same exact circumstances each time, only changing some parts of the configuration, such as the database template, the name of the environment, and which filesets to use. But since Kubernetes is declarative, there needs to be resource manifests for each environment. This is not something that can be manually templated for each environment due to its being too cumbersome.

However, a way to achieve per-environment resource templating is through the use of the Kubernetes package manager Helm [27]. Helm charts package all the required Kubernetes resources in a parametrized form, allowing environment-specific configuration to be injected at installation time using a "values" file. This fits our workflow well, where the small JSON file that is generated by the Web UI can be directly mapped to the "values" structure on a Helm chart. Compared to alternatives like Kustomize overlays or manually generated YAML, Helm offers stronger parametrization and a built-in integration with Argo CD. This makes Helm a practical way to template these short-lived and reproducible environments.

But the values.yml file still needs to be generated from user input. Contemporary research, such as Shrestha et al., tells us that GitOps offers advantages due to its automatic workflows for rollback and issue resolution [7]. Since we want to automate the workflows, we chose to use GitOps for our cluster bootstrapping. One of the most popular ways to implement GitOps in a Kubernetes cluster is through Argo CD, which is an open-source GitOps continuous delivery tool for Kubernetes [28].

The Argo CD ApplicationSet supports several generator strategies. For our use case, the primary design decision was between discovery-based approaches (e.g., SCM/Pull Request generators) and file-driven generation (Git generator).

We selected the Git generator because our workflow is explicitly request-driven: users create, update, or delete one environment by changing one configuration file. Other generators were considered, but they did not match this control model as well, either because they are static, combinatory, or oriented toward repository discovery rather than file-level environment requests.

The Git generator matches our requirements perfectly since it allows us to manage the Applications by creating and deleting files in the repository on demand, which is easy to do from the GitHub API. The approach also preserves the GitOps model by keeping Git as the single source of truth and naturally provides auditability and governance through the commit history of the repository.

There are other solutions to deploying Kubernetes resources onto a cluster, such as using Kubernetes Operators, which is a way to extend Kubernetes so that it creates a new resource type that simplifies application management [29]. The reason that we did not use Operators for this deployment is that they require significantly more engineering effort due to their having more implementation requirements, such as controllers, CRDs, and role-based access control (RBAC). Another reason is that they work best for long-lived applications that need continuous reconciliation, while our use case calls for short-lived applications that do not change during their lifetime.

The values from Listing 1 are mapped to the Helm values of the new Application. The `config.json` file is a JSON structured file that carries information both for the Web UI as well as for the Application itself. This follows the examples from the Argo CD documentation for Git generators [30]. This pattern lets us extract the values from the JSON using their keys like `{.environment.name}` and `{.templates.database}`.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: ifacts-applicationset
  namespace: argocd
spec:
  generators:
  - git:
      repoURL: https://github.com/iFacts/argo
      files:
      - path: "internal/application-config/**/config.json"
  template:
    metadata:
      name: '{{.environment.name}}'
    spec:
      source:
        chart: charts/ifacts-product
        helm:
          valuesObject:
            image:
              tag: "{{ .environment.hash }}"
            mssql:
              databaseTemplate: "{{ .templates.database }}"
              db_rev: "{{ .templates.db_rev }}"
            filestore:
              name: "{{ .templates.filestore }}"
        repoURL: harbor.kube.internalifacts.se
      project: "default"
      destination:
        server: https://kubernetes.default.svc
        namespace: playground
```

Listing 2: An example of a ApplicationSet manifest

```yaml
image:
  tag: sha-c936584

mssql:
  databaseTemplate: regression-insurance-minimal
  db_rev: 1

filestore:
  name: regression-insurance
```

Listing 3: An example of a Helm values.yml override file generated from custom configuration

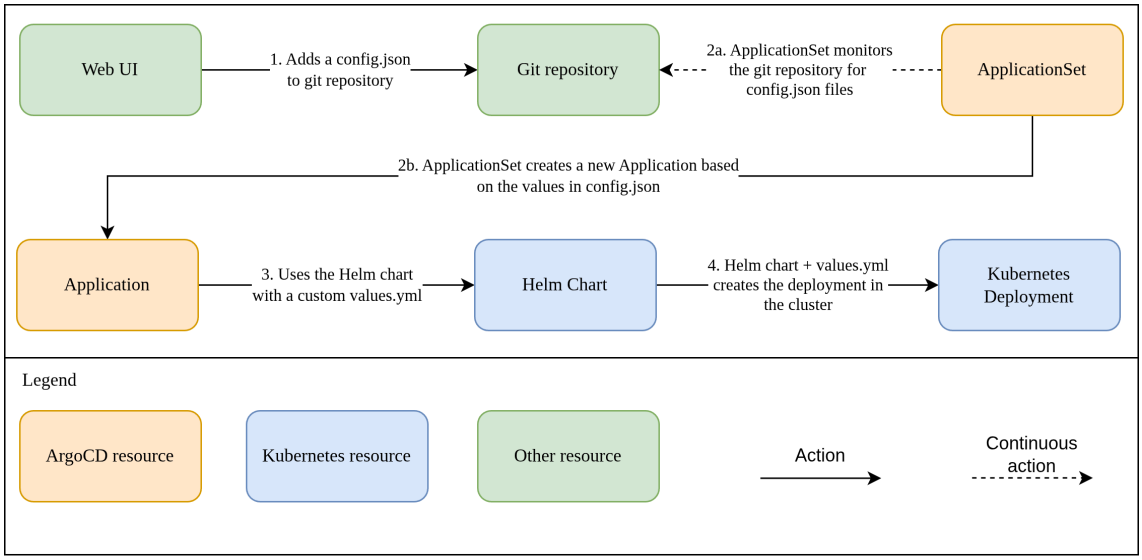

Figure 4: An overview of the process for templating the resources

The JSON files in the GitHub repository are made into EPEs by using Kubernetes in combination with Argo CD. The repository referenced earlier, where the Web UI creates, modifies, and deletes JSON files, is dedicated to Argo CD, and it is continuously monitored by it for updates. Each time Argo CD polls GitHub for changes to that repository, it will look for changes to every file in the specified paths within the repository. This means that the state of the GitHub repository is now the single source of truth for the target state of the environments.

4.4.1 Customer Dataset Templates

Since the EPEs are meant to be used as staging environments for different projects within the company, the user needs to be able to select which dataset the environment should use. This is crucial due to the fact that the software that the case company delivers is highly configurable [14]. Forcing the user to select a dataset ensures that the preview environment reflects the customer environment that the feature should be tested in, which would not be possible with a shared, generic dataset.

These datasets are anonymized copies of either shared test environments or production environments. In Figure 5, you can see that there are also different sizes for the databases. The smaller datasets are used for rapid deployments and fast functional testing, while the larger datasets support performance testing and the testing of issues that only occur in environments with production-scale database volumes.

To ensure consistency for all the datasets, they are stored as compressed SQL Server backup files. Using a backup file provides a snapshot in time for a database and allows for a deterministic restoration without any additional migration steps or complex schemas. The backups are stored in an S3-compatible object storage system, which provides a scalable, immutable, and versionable storage. This solution also allows for faster transfers from the storage to the cluster, as all the transfers happen in the cluster.

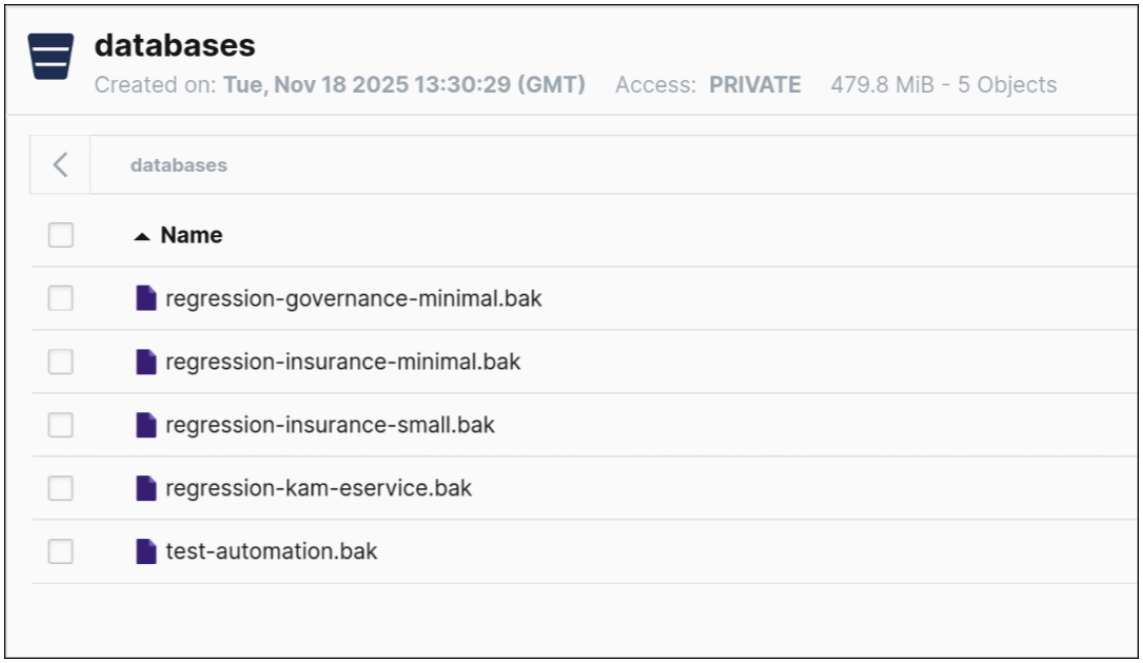

Figure 5: An image of the minimally viable templates that were used

The templates are also available for the developers if they want to download them for use in their local developer environments. This ensures that the local development, preview environments, and shared testing environments are all based on the same dataset versions, which improves reproducibility and reduces configuration drift across the development workflow.

4.5 Preview-Enabled Delivery Workflow

The technical solution in itself may not give any immediate benefit to the organization. To provide benefit, there needs to exist an established process on how to, when to, and why to use the EPEs. The proposed new process centers on improving the quality of the code in the "trunk" branch. The company has ambitions to move over to trunk-based development, but they plan to do so incrementally. Therefore, we suggest this process as a step in the right direction.

We propose implementing a pre-merge validation gate, where developers are required to create an EPE before merging the code into the trunk. This ensures that every code change is deployed and verified in a production-like EPE before even reaching the trunk branch. Our proposed solution relies on adding the EPE validation to the existing pre-merge checklist, as opposed to having it created automatically. This was decided due to the high cost of having an environment for every small feature running on the cluster. The cost of the compute that is needed is too high for the case company right now, but in the future, we recommend that they add it to their CI pipeline so that it gets created and verified for each PR.

The EPE request should be created manually by either the developer or the tester from the EPE portal; see Section 4.3 for more details on how the portal itself works.

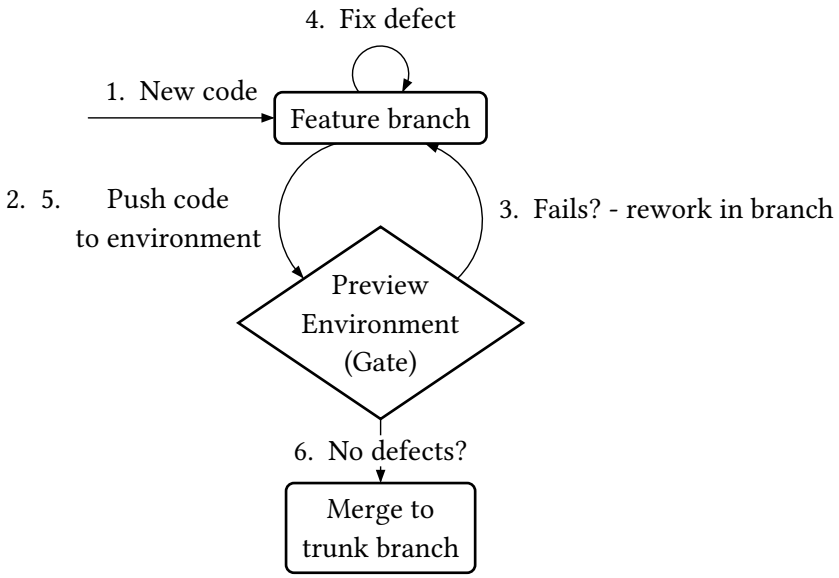

Figure 6: A figure of the proposed preview-enabled workflow

When creating the environment, the developer needs to choose which database template the environment should use. We chose to enforce manual selection since only the developers themselves know which database type or customer case they are working with. They then select their branch and a suitable TTL for the timeframe of the work item.

After the developer has created the environment, they validate that it works by first checking the status field in the portal and then by going to the UI of the EPE and verifying that it is working.

After the EPE is running, the developer sends the URL of the EPE to the assigned tester and stakeholder. Both the tester and the stakeholder prepare themselves for the micro-demo by going through the EPE and verifying the features. This helps them be prepared for the micro-demo session, where they can present their findings. During the micro-demo, the developers show the newly developed feature, and the tester and stakeholder present their findings if they have any. All defects are logged in the Jira story so that the developer has them documented.

When the developers have time, they fix the documented defects in the branch and commit the changes. They then go into the Web UI and click on the “update environment” button as is detailed in Section 4.3. This patches the config.json with the new commit, causing Argo CD to redeploy the EPE with the latest code.

When the EPE is updated, the stakeholder needs to sign off that the environment is free of defects. The criteria for passing the gate are that the EPE is in a healthy state, the feature works with the specified dataset, and all acceptance criteria defined in the story must be verified by both the developer and tester. If all of these criteria are met, then the stakeholder signs off on it. After passing this gate, the branch can be merged with the trunk branch. This cycle can be seen in Figure 6.

After the branch has been merged, the developer should go to the UI and delete the EPE manually. An alternative to this is to have a webhook that is set up on PR merge that deletes the EPE. This has a considerable engineering overhead and was out of scope for this thesis.

By enforcing this workflow, the organization takes a step in the right direction towards their goals of moving towards Trunk-based development (TBD), where developers commit directly to the main branch. This workflow seems to be most suitable for smaller and more experienced

teams, which suits iFACTS perfectly since the development team is 15 experienced developers [1].

The workflow itself is not officially rolled out, so we cannot measure anything in the field. However, in Section 5, we evaluate a handful of features that use this gate. We measure the workflow's effectiveness by how much rework time can be avoided by following the workflow.

Chapter 5
Evaluation

In this section, we evaluate the solution from Chapter 4. We start by detailing the methodology of our evaluation in Section 5.1. After this, we detail what data will be collected in Section 5.2, followed by how we analyze it in Section 5.3. Following this, we present the results in Section 5.4, and finally the limitations of the evaluation in Section 5.5.

5.1 Research Design & Methodology

To assess both the effectiveness of the EPE workflow (RQ4) and the likelihood of stakeholders using the new workflow (RQ3), we adopted an embedded case study design following the guidelines of Runeson and Höst [31]. An embedded case study examines a single case containing multiple units of analysis, where the case in our situation is the organization and the units are the teams and processes. This is opposed to a holistic case study that treats the case as a single, indivisible unit of analysis.

Since we are using multiple units of analysis, both adoption and effort reduction, it is more appropriate to use an embedded design rather than a holistic design. This also enables us to use a mixed-methods design, since RQ3 requires qualitative data while RQ4 requires quantitative data.

We chose to run a case study because of its appropriateness in evaluating workflows in real organizations. An alternative would be a controlled experiment, which would be hard to run due to our not being able to randomize whole teams or sprints because of the smaller size of the company. Other constraints to controlled experiments are that there was a limited number of releases during the thesis timeframe, which made it hard to compare release to release. A controlled experiment would also mean that defects would have to be shipped to the main branch, which has implications for the code health of the main branch as well as ethical implications on the company standards.

The units of measurement will differ for each research question. RQ3 will use the interview data after a subject has provisioned and performed a set of tasks concerning an EPE. Using interviews together with TAM constructs is a classic way of capturing the adoption of technological solutions. RQ4 will be evaluated using an estimation of the cost difference between solving issues found before and after merging to the trunk. We consider estimated time and effort to be an appropriate proxy for the actual cost, due to the very direct effect that billable time is equivalent to money.

We utilize a Goal-Question-Metric (GQM) plan to keep the evaluation traceable to the research questions [32].

Goal	Question	Metric
Estimate willingness to use the suggested workflow pre-merge feature validation (RQ3)	What is the stakeholder's perception of the usefulness and ease of use of the new workflow?	Perceived Usefulness (PU) and Perceived Ease of Use (PEOU) based on the Technology Acceptance Model [10]
Estimate rework effort avoided by using EPEs (RQ4)	What is the difference between the rework effort in the EPE and the predicted effort if the defect leaked to the trunk?	Three-point estimation, diagnosing and fixing before merging to trunk vs. trunk rework effort [33].

Table 2: A table listing the Goal Question Metric (GQM) data

Using the GQM approach in Table 2 ensures that we can trace back each collected metric to a specific question and, in turn, to our high-level evaluation goals, thereby reducing the risk of collecting irrelevant data.

5.2 Data Collection

To address the likelihood of adoption, we will conduct sessions with potential adopters, where they will test the usability of the Web UI, answer questions about the usability and trust in the workflow, in order to map the answers to the TAM constructs mentioned in Table 2. The willingness to adopt a new workflow is a complex thing to measure, due to its socio-technical nature. One key factor in the willingness to adopt a workflow is its perceived usability. This is supported by the definition of usability, which states that a low perceived usability is correlated to unwillingness to use the system [34].

5.2.1 RQ3 - Perceived Usefulness

Since they are intended users of the system, selected participants are the developers and members of the Quality Assurance (QA) team. It was important to have both members of the

QA team as well as developers as participants due to the fact that they will interact with the workflow differently. Developers will mostly use the workflow to create environments and validate their code changes, while the QA team members will interact with the workflow in more detail to update the environments to new versions. The members of QA can also give us insight into how this will actually affect their testing work.

All members of these teams are deemed capable enough to give relevant and reliable data. This made the selection of subjects mainly dependent on availability during the data collection period and equal representation between the groups. Since we are aiming for analytic insight in the adoption of the workflow and not statistical generation, it was deemed that this pragmatic sampling was justified. Data collection continued within the constraints of the time frame, resulting in a sample size of four subjects, one developer, and three QA members.

Task Number	Task Description
1	Create an EPE from a specific commit with a specific time to live date.
2	Extend the EPEs time to live.
3	Change the commit from which the EPE is created.
4	Reset the EPEs database.
5	Opening the actual EPE.
6	Delete the EPE.

Table 3: The list of tasks that the participants were asked to perform

During the process, we will utilize a *Think-aloud* protocol, which is when the participant is asked to voice their immediate thoughts as they use the tool. This protocol is suitable for these types of usability studies since it often captures usability issues that standard interviews can miss [35].

At first, the subjects will be briefed about the new workflow and EPEs, since the concept is new to some. The participants will be recorded while performing a set of tasks, listed in Table 3, in the Web UI. The tasks are chosen to cover the full EPE lifecycle, from creation and management to deletion.

5.2.2 RQ4 - Rework Effort Estimations

Data collection for this stage will be based on specific stories that use micro-demos. A scenario where a micro-demo would normally use a development environment will, in this case, use an EPE instead. Micro-demos were chosen as an evaluation unit because they represent a realistic validation point where defects would typically be discovered before being merged.

Which stories to use will be picked based on convenience sampling, based on developer availability, the released schedule, and the help of a Business Analyst who helped pick sufficiently complex stories. Convenience sampling was primarily picked due to the right release schedule at iFACTS and the minimal availability of micro-demos to pick from at this stage of the sprint. Given the exploratory nature of RQ4, convenience sampling was considered appropriate.

In this scenario, a *defect* is defined as any type of issue that will require a developer to edit the code. It can be a bug, leaving the system unusable, or a minor visual glitch in the UI.

5.3 Data Analysis Methods

The objective of RQ4 is to quantify the cost savings obtained by detecting issues while using the EPEs in the suggested workflow, as this will prevent the issues from leaking into the trunk branch. The cost savings are quantified using time spent on rework as a proxy. This is a common practice in software engineering studies, as it directly reflects developer time spent on resolving defects.

5.3.1 Workflow Adoption and Usefulness

To evaluate the system's acceptance, a semi-structured interview with open-ended questions will be held after the tasks have been performed. The interview questions were based on standard TAM questionnaire questions, but modified and tailored to our solution. Semi-structured interviews were used to allow for unexpected concerns to surface, such as comments on the old workflow compared to the new workflow.

To ensure consistency across interviews, a shared interview guide was used. This guide was followed consistently across all interviews to ensure internal validity. The guide consisted of starting with asking the participant for consent to be recorded, as well as that their identity would be anonymized. After this, the interview questions were asked in order. However, since the interview was exploratory, we allowed for spontaneous follow-up questions to be interjected.

Question	TAM-construct
How did the EPE compare to a production context? If there were any gaps, what mattered most in terms of configuration or data?	Perceived Usefulness
How much confidence do you have in finding bugs or defects in the EPEs compared to using your old environment workflow?	Perceived Usefulness
How did using the EPEs compare in time and effort to your previous workflow?	Perceived Usefulness
Would the suggested workflow change your job effort to the better or worse in any way? If so, how?	Perceived Usefulness
How intuitive did you find your first setup of an EPE with the new system?	Perceived Ease of Use
Did you find it hard to set up your first EPE?	Perceived Ease of Use
Are there any specific factors or friction points that would prevent you from adopting the workflow?	Perceived Ease of Use

Table 4: The list of post-demo interview questions with their mapped TAM constructs

The questions are mapped with Technology Acceptance Model (TAM) constructs, specifically shown in Table 4 [10]. The constructs that the questions are mapped to are Perceived Usefulness (PU), which is the degree to which the user believes using the EPE will improve their job

performance. As well as the Perceived Ease of Use (PEOU), which is the degree to which using the new workflow is free from effort. The mapping from question to construct was derived from the questionnaire that was used to create the interview questions.

The motivation for using TAM constructs such as PU and PEOU is that this model was explicitly designed to measure initial workflow adoption, and they are especially well-suited for pilot studies because they do not require extensive usage history. Other methods for measuring benefits to workflows, such as other information systems success models, rely on use already having occurred [36]. Since the scope of our thesis did not allow us to introduce the tool, we could not assume its use during the evaluation.

Another similar model to TAM that could have been used is the *Unified theory of acceptance and use of technology* (UTAUT) by Venkatesh et al. It extends TAM by moderating variables, incorporating social influence, and facilitating other conditions. However, it also assumes an existing organizational embedding as well as available infrastructure and support. Given the exploratory nature of early workflow adoption with the absence of stabilized organizational conditions, TAM offers a smaller yet theoretically grounded alternative to UTAUT.

Moreover, observational data from the recording, including data from the Think-aloud protocol, will be used to triangulate information and add data to the PEOU. For example, if a subject claims that they found the system to be easy to use but struggled or hesitated with specific parts during the session, it will be taken into account. By triangulating the information, we aim to reduce self-report bias and thereby improve construct validity.

Through deductive coding, phrases for each answer from the interviews will be categorized into either positive, negative, or neutral value for each construct. The initial set of codes was defined in advance, while allowing for a limited amount of inductive refinement as transcripts were analyzed.

Quantifying the results from the codes and transcripts into concrete mappings to the constructs will give us insights into whether or not the stakeholders are likely to accept the suggested workflow or not [10]. Using coding allows us to preserve the theoretical grounding of the interviews. And by quantifying, we can compare results across participants and find patterns in the answers of the different participants from the QA team and the Dev team.

5.3.2 PERT

When evaluating rework effort for the time spent on resolving a defect, either pre-merge or post-merge, there were two main alternatives. The first was to use estimation, where we ask the developer to give us an estimate of how much time resolving the defect would take, pre-merge and post-merge. The other alternative would be to observe and record how the developer solves the issue in the different scenarios.

Estimation was chosen since post-merge effort cannot be measured and observed, since this would mean that we would have to let defects into the trunk voluntarily, which would be an ethical concern, as we would explicitly deteriorate the quality of the trunk.

For every defect that was found, the developer was asked to make an estimate for the amount of time it will take to solve it using three estimates: the best case o, the most likely m, and the worst case p. These estimates were made for both the *pre-merge* scenario, where the defect is resolved before being merged, as well as the *post-merge* scenario, where the defect is detected after merging to the trunk and has to go through the entire SDLC cycle. Using the same developer

for both estimates reduces variability caused by confounding factors such as differing familiarity with the code and differing competence in that area of the codebase.

We use Program Evaluation and Review Technique (PERT) to weigh the values into an average value [33]. This is done to mitigate uncertainty and biases since estimates in software engineering are notoriously prone to optimism biases, where developers fail to account for unforeseen circumstances [37].

$$E = \frac{o \quad + \quad 4m \quad + \quad p}{6} \tag{1}$$

See Equation 1 for how the calculation for the weighted average E was calculated. The quantifiable benefit of the actual switch can then be determined by the difference ΔE in Equation 2.

$$\Delta E = E_{\text{Post-merge}} - E_{\text{Pre-merge}} \tag{2}$$

If the micro-demo was a *clean run*, meaning no defects were found, it would result in a $\Delta E = 0$, and it would still be taken into account when calculating the final average. As excluding them would result in a selection bias and would artificially inflate the averages by implying that every micro-demo would result in defects being found.

The results will be presented as tables for the ΔE detailing the calculations for each story. Where ΔE indicates the time saved by using an EPE for the Micro Demo.

$$\Delta E_{\text{dec}} = \frac{E_{\text{post}} - E_{\text{pre}}}{E_{\text{post}}} \tag{3}$$

From these values, we can then calculate the average relative decrease in effort.

$$\overline{\Delta E_{\text{dec}}} = \frac{1}{x} \sum_{n=1}^{x} \Delta E_{\text{dec}} \tag{4}$$

Finally, a relative decrease average $\overline{\Delta E_{\text{dec}}}$ is calculated across all micro-demos using Equation 4. Aggregating all individual ΔE_{dec} will determine an average potential cost saving. These results are indicative of potential cost savings rather than being causal evidence.

5.4 Results

We will start by presenting the results where we triangulated the results from the deductive coding of the post-session interviews with the Think-aloud protocol. We will then map these

results to the TAM constructs *Perceived Usefulness* and *Perceived Ease of Use*. Finally, we go on to present the PERT estimations given by the developers.

We interviewed two members of the QA team and two developers for this part of the evaluation. These participants are marked as QA-member 1 and 2, as well as Developer 1 and 2.

5.4.1 RQ3 - Perceived Usefulness

All interview transcripts were coded using deductive coding as per Section 5.3.1. The codes were then marked by their sentiment, with either being positive, negative, or neutral. The frequency of each sentiment was then recorded in Table 5.

Construct	Positive Codes	Neutral Codes	Negative Codes
Perceived Usefulness (PU)	7	2	1

Table 5: A table detailing the frequency of positive, neutral, and negative coded statements regarding Perceived Usefulness

From the codes we could extract broader themes related to the contents. We identified four key themes related to the perceived usefulness: *Efficiency and Time Management*, *Data Integrity and Confidence*, *Configuration Control*, and *Workflow Compatibility*.

5.4.1.1 Efficiency and Time Management

The new workflow's most prominent theme in the interviews was a significant reduction in setup time. However, the perspective changed depending on whether the interviewee was a developer or a member of the QA team.

While the QA-team noted that they were able to bypass an entire step in the process, when asked how the EPEs compare in time and effort compared to their previous workflow, one member from the QA-team responded with:

> it's much quicker. For sure. Um, setting up the usual static environments takes… takes quite some time. Um, so that's nice… differences are maybe that it's a bit to reboot them… But otherwise it's just a time… time-saving way.
>
> — QA-Member 1

The developer's focus was directed towards the feedback loop between coding and QA verification (where they use staging environment) instead of setting up the environment. They highlighted that they often wasted time debugging environments that had not been updated with the latest code changes yet. Which could no longer happen in the new workflow.

> Way faster. It's way faster. Because before, well, I was merging the PR, and then sometimes the QA team came back to me and they were like, 'Hey, the bug is still there.' And after maybe half an hour of checking, I would realize that the environment was not updated with my changes, you know?
>
> — Developer

5.4.1.2 Confidence in the System

The confidence that the system would reliably be able to identify defects was lifted as a central theme in the interviews. Comparing their previous workflows, both developers and the QA team expressed increased confidence in the system, albeit the source of confidence differed depending on the role.

For developers, the confidence boost was primarily derived from the fact that they are able to test their environments against production-like datasets. Thus, eliminates the uncertainty of local-only configurations before it leaves the feature branch. One developer highlighted how this removes the ambiguity of their local developer environments:

> Well, I'm very confident in using it because on my local machine, I might have some faulty database. In the ephemeral environment, I know for sure we're using a database that's being used for regression testing, which gives me a very high confidence. Yes, so that's it pretty much. 100%.
>
> — Developer

The largest confidence boost for the QA team was that their environments always start from a clean slate, thus minimizing potential configuration drift.

> Hmm. I sort of actually think it would be better. Like, easier to find bugs. Because sometimes in our development environments, we have a lot of old configuration [sic] and trash data that sometimes cause issues that aren't real issues. But since you redeploy the database and everything here, it might actually remove that issue.
>
> — QA-Member 2

However, one of the QA members raised an important issue regarding the datasets, which potentially lowers the confidence levels. The EPEs rely on templates created based on real customer data; these also run the risk of being outdated and are prone to configuration drift. If the developers and QA team can not trust them to be up to date, it would lower the confidence level in the system and, in turn, the PU.

> I would say that it's depending [sic] a lot on which database templates there are... like available. Um, but also since they are database templates, they I assume that they will, by nature, get outdated.
>
> — QA-Member 2

5.4.1.3 System Configurability and Features

Although only noted by one of the QA members, one important theme was the lack of certain features. Even if they are not needed often, leaving out features might cause critical blows to the PU, which was quantified as a negative code in the PU.

> I think it's good. Yeah. Um, it's pretty clear what you can do. Um, there might be a need to have some way of restarting an environment from the UI. Since that becomes a bit tricky. And also to know which feature flags your environment will sort of ship with. Since that's also hidden from you.
>
> — QA-Member 2

5.4.2 RQ3 - Perceived Ease of Use

All interview transcripts were coded using deductive coding as per Section 5.3.1. The codes were then marked by their sentiment, with either being positive, negative, or neutral. These frequency of each sentiment were then recorded in Table 6.

Construct	Positive Codes	Neutral Codes	Negative Codes
Perceived Ease of Use (PEOU)	5	3	0

Table 6: A table detailing the frequency of positive, neutral, and negative coded statements regarding PEOU

From the codes, we could then extract more general themes. We identified two central themes: that being *Setup and Learnability* and *UI Frictions*, which mostly relate to the Web UI.

5.4.2.1 Setup and Learnability

The most common theme concerning the easy setting up of a new EPE, which for both developers and the QA team means mostly using the Web UI to create the EPE, in addition to doing some configuration. For all subjects, there was a clear overall positive experience in these aspects. No major usability issues or hesitations occurred during the Think Aloud sessions, and both Quote 7 and Quote 8 represent well what was said about the setup during the interviews and how setting up EPEs went during the sessions.

> It was really easy to use… very user friendly… just a couple of small details: giving a name… selecting a branch… selecting an existing database… time to live and that's it.
>
> — Developer 2

Regarding learnability, the general opinion was that the workflow would be quick and easy to master. One of the QA members estimated that the adoption of the new system would be almost immediate.

> I think... I think it would take like a day for us to get used to it.
>
> — QA-Member 1

5.4.2.2 UI Frictions

Although most of the tasks went well for the subjects, it was less obvious for some of them, as mentioned in Quote 8. This subject was familiar with all the icons and had no problem with any of them. Even if it was not mentioned during the interviews, the Think Aloud sessions clearly showed some struggles to understand certain parts of the tasks. We noted two hiccups during the Think Aloud sessions, which were not mentioned during the interviews, which we account for when calculating the results for the PEOU.

Two users struggled to find the button for changing the commit; they did not recognize the branch (a standard git-branch symbol) icon as a possible identifier for it, and instead used the exclusion method to identify which buttons to exclude. This caused the process to take about ten seconds extra. This was also a potential problem identified by a user who did identify the button successfully Quote 9.

One user also struggled to enter the Time To Live at first and tried to manually enter the date instead of using the Date Picker. They also struggled to close the window after they had picked the date since it did not close automatically.

Although these friction points occurred, a majority of the tasks were completed without hiccups, and the user experience was positive according to the interviews. Therefore, these instances were coded as Neutral, as opposed to Negative.

> The only thing would be if someone without maybe much insight into Git... how they would know which commit to use... the pull request can become sort of half famous... the commit itself might not be as known.
>
> — QA-Member 2

5.4.3 RQ4 - Rework Effort Estimations

During the Micro-Demos, three defects of different sizes were found, one for each Micro-Demo. Data was provided by each responsible developer, according to the strategy described in Section 5.3.2.

Story	Scenario	Optimistic (h)	Most Likely (h)	Pessimistic (h)	Expected Time (h)
Defect 1	Pre-merge	1	1.5	3	1.67
Defect 1	Post-merge	1.5	2	4	2.25
Defect 2	Pre-merge	1	3	4	2.83
Defect 2	Post-merge	4	8	17	8.83
Defect 3	Pre-merge	3	8	16	8.5
Defect 3	Post-merge	8	17	40	19.3

Table 7: A table detailing the results for the Three Point Estimations

As shown in the Table 7, the expected time to fix defects in the post-merge state was consistently higher than in the pre-merge state. Although to a varying degree, depending on the defect.

The quantifiable benefit for each defect was calculated as the relative decrease in rework effort (ΔE_{dec}), see Equation 3 for details. A value of $\Delta E_{\text{dec}} = 1.0$ corresponds to a 100% decrease in effort in resolving the issue pre-merge as opposed to post-merge.

Story	ΔE_{dec}
Defect 1	0.257
Defect 2	0.679
Defect 3	0.559

Table 8: A table detailing the results for the relative decrease in rework effort per defect

Finally, the average across all of the defects was calculated as per Equation 4.

$$\overline{\Delta E_{\text{dec}}} = 0.4989 \tag{5}$$

This results in an average relative decrease of 49.89% in effort per issue when using post-merge testing as opposed to pre-merge testing.

Using data from internal time reports, we could see that the company collectively spent 2170 developer hours and 716 QA hours on handling internal bugs from Q1 to Q3. If our results were relevant for all of these bugs, it would result in a time savings of:

$$\begin{aligned} \text{Time saved} &= E_{\text{post}} - E_{\text{pre}} = E_{\text{post}} - \left(E_{\text{post}} - \overline{E_{\text{dec}}} \times E_{\text{post}}\right) \\ &= -\overline{E_{\text{dec}}} \times E_{\text{post}} = -0.4989 \times 2886 = -1439.8 \text{ hours} \end{aligned} \tag{6}$$

This results in a total time saved of approximately 1440 hours for the three quarters of the working year. The extrapolation is, however, indicative and assumes similar defect character-

istics and a consistent use of the validation workflow for all of the internal bugs from the time reports.

5.5 Limitations of the Evaluation

To ensure transparency in the interpretation of the results, it is necessary to acknowledge the limitations of the evaluation.

5.5.1 Scope and Sample

The primary limitation regarding data collection for the evaluation for both RQ3 and RQ4 is the small sample sizes. It removes the ability for the result to stand alone as proof and should instead only be regarded as an indication within this specific team.

Furthermore, the sampling strategy was pragmatic and based on availability. This could add a bias since there is a definite possibility that a certain type of group will be available at the same time. However, not being able to pick subjects ourselves also removes the ability to accidentally pick subjects we deem extra suitable for our study.

5.5.2 Case Study

As this was a case study within a single organization, there are factors that may be very specific to this particular company, which might not be applicable to other companies. For example, the workflow suggested was structured to be as similar as possible to the one already used in iFACTS to help with the ease of use.

5.5.3 Exposure Time

Both the Think-aloud session and Micro-Demo are fairly short interactions for the subject to have with the Web UI and the workflow, and are far from sufficient to get the big picture of how it will be to work with in the long term.

5.5.4 Estimation Reliability

There is a significant reliance on the developers' ability to be subjective and make good estimations for the pre- and post-merge scenarios. While PERT is used to mitigate optimism bias, it is impossible to fully eliminate the inherent uncertainty of estimations.

5.5.5 Social Desirability and Researcher Bias

The evaluation was conducted within the researcher's own organization and involved colleagues; there is a certain risk for the participants to subconsciously give more favorable responses or withhold criticism.

Chapter 6

Reflection

This chapter situates our findings by revisiting each research question with the findings from Chapter 5 in mind. We then analyze the strengths and limitations of the methods that we chose. After this, we discuss threats to the validity of both the evaluation and the newly suggested process. Finally, we suggest future work in the field based on the lessons learned during both the implementation and evaluation.

6.1 Situating our Findings

This section situates the findings by combining insights across the research questions and reflecting on their implications for the new workflow.

6.1.1 Revisiting Our Problem Framing

By revisiting the initial problem framing, we can see that the evaluation largely confirmed the identified limitations of the original workflow. Challenges like the replication of environments, late defect detection, and issues with sharing an environment were all repeatedly mentioned in participant feedback, highlighting that the problem formulations captured core constraints faced by QA and developers.

However, the evaluation also highlighted that some issues, such as branch management complexity, were a bigger problem for QA than for developers. This points to the role-dependent nature of the problem statement, where the solution needs to be tailored to the needs of both teams.

6.1.2 Implications on the Evaluation of the New Workflow

The results from Chapter 5 suggest that the proposed workflow alters how validation activities are distributed across development and testing roles.

Developers perceived the ability to validate changes in production-like environments prior to merging to be useful. This allows them to focus on finishing tasks completely before moving on to new tasks.

Participants from the QA team emphasized how the gained environment isolation and state reset were beneficial to them when testing features. This enabled a workflow that was more focused and allowed them to feel less constrained when working with data in the environments.

These role-specific perceptions highlight that the perceived usefulness of the workflow is closely tied to existing responsibilities and pain points.

6.1.3 Implications of the Rework Avoided

The effort estimates indicate that the introduction of ephemeral environments may shift, rather than uniformly reduce, development and testing effort. While certain validation activities may require additional setup time, this effort is offset by earlier defect detection and reduced rework at later stages.

The observed variability in estimates further suggests that the impact of the workflow is sensitive to feature complexity and timing within the development lifecycle.

6.1.4 Reflection on Using TAM as an Analytical Lens

The Technology Acceptance Model provided a good framework for interpreting how the participants perceived the workflow, especially with respect to the perceived usefulness. We observed many themes such as increased confidence, improved efficiency, and greater flexibility in cherry-picking commits to environments, which all aligned well with the TAM constructs.

However, there were also several findings related to data management, environment isolation, and organization dependencies that could not be captured by individual perceptions. These findings highlight concerns at the process and infrastructure level. This suggests that TAM is efficient and effective at capturing individual user acceptance, but it may be insufficient in capturing the acceptance of these workflow-oriented DevOps tools.

6.1.5 Overall Implications for Ephemeral Environment Provisioning

Taken together, the findings suggest that ephemeral environment provisioning can support development as well as QA workflows by enabling earlier and more isolated validation, together with efficient branching solutions where commits can be cherry-picked to the environment. While the first two research questions served to ground the design of the solution, RQ3 and RQ4

provide empirical evidence and insight into how the workflow is experienced by the employees using it.

However, the benefits of such an approach are dependent on solid automation and representative data management, as well as commitment from the organization to maintain the needed infrastructure. Without these key tools, the added benefits might be outweighed by the uncertainty of the tools and the lack of data.

6.2 Validity of Evaluation

This section details key threats to the validity of the evaluation and describes how these were mitigated, along with remaining limitations. We divided these into different groups: the validity of the constructs, the internal validity of the evaluation, the external validity of the evaluation, and finally, threats to the reliability of the evaluation.

6.2.1 Construct Validity

The validity of the constructs is solidified by the operationalization of perceived usefulness and perceived ease of use using well-established TAM constructs.

6.2.2 Internal Validity

One internal validity threat is the threat of the novelty effect, which is a phenomenon in which new workflows have a positive influence on perception. This threat was possible for the participants due to the fact that it was the first time using the tool for many of them. To reduce the risk of this effect, the participants were explicitly encouraged to compare the new workflow against their current practices during the interview.

Another threat to the internal validity is facilitation bias, as the researchers were present during the sessions and introduced the workflow to the participants. This bias was mitigated by using a task list that was defined beforehand and by following the same interview guide across all participants. It was also mitigated by allowing the participants to express negative feedback at any time, and the negative feedback was even encouraged. Nonetheless, the findings may still reflect short-term perceptions formed in the controlled environment as opposed to actual stabilized usage behavior.

6.2.3 External Validity

The external validity is limited by the fact that the evaluation was done in a single-company setting with a relatively small number of participants. Our findings, therefore, cannot be statistically generalized to all kinds of development contexts or organizations.

However, having both developers and QA roles present during evaluation supports analytic generalization by capturing perspectives from the primary stakeholders involved in the measured activities, such as environment provisioning and testing. Our findings are applicable to

organizations with similar characteristics, such as an established CI/CD pipeline, containerized infrastructure, and, most importantly, a need to support customer-specific software variants.

Transferability may be higher for teams that are facing similar challenges as the case company, such as using a shared staging environment and having a lack of shift-left testing practices currently.

6.2.4 Reliability

The reliability of our evaluation is supported by using a shared task list and interview guide consistently across all sessions. This ensured that the interview participants were subjected to the same conditions, which increases internal validity and reliability. When coding the interview transcripts, we grounded the process in predefined TAM constructs to improve transparency and traceability of deduced codes. However, despite these efforts, there is always a risk of residual subjectivity remaining when interpreting qualitative data.

6.3 Validity of Suggested Process

This section discusses how boundary decisions and assumptions affect whether the suggested new workflow can be applied successfully in practice at the company.

6.3.1 Feasibility Assumptions and Risks

The new process assumes a high level of automation maturity in the areas mentioned in Chapter 4, such as Kubernetes, Argo CD, MinIO, database template management, Docker image building pipelines in GitHub, and more. The effectiveness of the process highly depends on these socio-technological factors.

If these assumptions are not met, there may be a risk of the environments using outdated databases, environments not building properly, and error log outputs not being found by users. This will cause users to lose trust in the system and process, which might cause them to abandon it.

One key threat concerns the upkeep of the database templates and managing an up-to-date catalog of them. Multiple interviews uncovered that to be able to test all different versions of the code, it was important to keep a catalog of different database templates that are also versioned to the version of the code that they should use. If the templates become outdated, it could likely affect the usefulness of the environment itself, as it would be very hard for the user to deduce why the database migration is failing.

To mitigate this concern, it is important to set up proper database template automation that syncs current staging databases to the template storage regularly. Another solution could also be to create tools that automatically anonymize customer databases and create database templates of these. This would greatly improve the quality of the templates, since they would be nearly production-like in their structure. However, this is a greater effort on the side of the company, and would require engineering overhead and work efforts in automatic database scripting.

Another key risk concerns the ownership of the administration of the new process. Sustained use of the new process requires that one or more employees take ownership of the tools and

processes. Without this clear ownership, the maintenance of the workflow might be ignored, which could lead to users of the system leaving it for their former workflows.

While general automation does mitigate many of the threats to the feasibility of the process, it also introduces engineering and maintenance costs. The validity of the new process, therefore, depends not only on the technical feasibility but also on whether the organization is willing to invest in the needed supporting infrastructure and engineering hours to support the process.

6.3.2 Failure Modes

Failure modes for the EPE provisioning workflow include when an environment points to a non-existent container image, when the networking configuration fails for any reason, and finally, when the database templates can not be accessed or restored to the database.

In these cases, recovery would require manual intervention on the part of the administrator of the EPEs. This implies the need for the aforementioned role of an EPE responsible, who will administrate the database templates, container images, and the networking of the cluster.

6.3.3 Edge Cases for EPEs

The proposed process may be less suitable in conditions when the application uses a lot of external service integrations. In these scenarios, a shared staging environment may remain preferable due to rate limiting on the service integration supplier side or a limited set of connections using the integration provider. However, if the integrations allow for a normal rate of connections and there are mock accounts that will not affect the real services in a destructive manner, then this may be worth implementing into the ephemeral environment templates.

6.4 Methodological Reflections

In retrospect, the choice of using an embedded case study design was appropriate for evaluating the new workflow within the context of a real organization. This design allowed the study to capture perceptions related to adoption and also implications related to effort using different units of analysis. These measures would have proved difficult to isolate in a holistic or an experimental design. However, the context of a case study did constrain the ability to observe long-term effects and also limited the number of evaluation instances. This highlights the trade-offs that had to be made between real ecological validity and generalizability.

The mixed-methods approach that was used in the evaluation allowed the study to use both the qualitative insights into the perceived usefulness and ease of use, together with quantitative estimates of rework effort. This provided a more complete view of the impact of the workflow. While the qualitative findings helped to contextualize the rework estimates, the integration between the different measures was largely interpretive rather than analytical. In future work, tighter integration between the methods could strengthen the connection between the perceived effects and the observed effects. An example would be through more rigorous measurements over time regarding rework avoided through the use of the ephemeral environments.

The use of the *Think-aloud* protocol was valuable for the qualitative part of the evaluation, by allowing us to reveal usability frictions that were not articulated during the interviews. Partici-

pants displayed signs of observed hesitation and also found workarounds during task execution, which highlighted minor interface issues as well as a mismatch for participants who were not fluent in the concepts of Git. However, the method of *Think-aloud* may also have influenced the behavior of the participants by encouraging a more deliberate interaction. In normal use, the participants might not have focused as intently as they did during the session. This suggests another trade-off between the amount of insight that we could gather and naturalistic use.

Finally, the deductive coding strategy for our TAM constructs provided us with a solid analytical structure that supported traceability between the foundational theory and the empirical findings. It allowed us to operationalize qualitative concepts and data that otherwise would have been hard to extract. However, this approach also introduced risks in constraining the analysis by focusing on predefined constructs instead of themes that emerged naturally. While some inductive refinement to the codes and themes was allowed, larger process-level and organizational concerns extend beyond the scope of a TAM. This highlights the trade-off between a solid theoretical grounding and a flexible analytical framework when applying acceptance models to tools.

6.5 Practical Implications and Design Recommendations

Index	Recommendation Description
1	Define ownership for the database templates
2	Add UI support for commit or PR selection and feature-flag visibility
3	Provide controls for safe reset or restart
4	Decide when to use EPEs in your process
5	Create on-boarding documentation for users with lower Git fluency

Table 9: A list of design recommendations for implementing a EPE process

Chapter 7
Conclusion

This thesis addressed a recurring problem in customer-specific software development: defects that only appear in production-like conditions often bypass local development setups and then surface late in shared staging. In our case study context, the central mechanism is replacing this late, shared-environment discovery with branch-scoped EPE validation before merge. Defects found late are expensive to reproduce due to shared state, configuration drift, and limited representation of customer data. The case organization clearly displayed these challenges, particularly where multiple customer variants had to be validated.

To address the problem, we pursued four research questions. Firstly, we identified pain points in the existing environment provisioning and validation workflows, with special attention to the differences in developer and QA practices. Secondly, we designed and implemented a GitOps- and Kubernetes-based ephemeral preview environment (EPE) platform using Helm, Argo CD, Kubernetes, and MinIO, complemented by versioned database templates. Thirdly, we proposed a new workflow for development using a pre-merge "preview gate" that shifts feature validation to the pre-merge stage. Finally, we evaluated the workflow from an adoption perspective using TAM constructs, as well as from an effort perspective, using PERT-based rework estimation. All taken together, these contributions combine new technical design with an empirical evaluation in a real organizational context.

The results of the evaluation indicate that developers and QA alike perceive the new workflow as a confidence booster in enabling faster validation of changes. Other valuable features were the environment isolation and the ability to reset the state; these were especially valuable for the QA team for reducing data fragility and coordination overhead during testing. Some usability frictions were noticed, especially around the selection of commits and the time-to-live configuration, but not in the core workflow itself. Importantly, our findings also revealed that the perceived usefulness is coupled to the freshness and representativeness of the database templates, as well as to the existence of clear ownership in maintaining the supporting infrastructure.

As for the effort estimations, the study revealed a mean avoided resolution effort of 5.5 hours per story across all evaluated stories. While indicative and not causal, these results indicate

that earlier defect detection using EPEs can offset the setup effort that is required by the workflow. However, the effectiveness of these benefits is highly dependent on the reliability of the automation and a sustained effort in the quality of the dataset. Without these efforts, the cost of maintenance for the workflow risks outweighing the avoided effort from an earlier validation.

There are several key limitations that affect the evaluation of this study. First, it was conducted as a case study at a single company, which means the results may be strongly context-specific. Second, the small number of participants and limited exposure time reduce the potential for statistical generalization. Third, the reliance on estimated rather than observed effort introduces uncertainty in the quantitative findings. Finally, because the study was conducted among colleagues, there is a potential for social desirability bias. Taken together, these limitations suggest that the results should be interpreted as exploratory rather than analytically generalizable.

Future work should first and foremost focus on a technical standpoint to evolve the workflow while focusing on the problems highlighted by subjects in the interviews and observations in the Think-aloud sessions. This would include expanding the features of the Web UI with feature flag visibility and restart buttons for the environments. A few improvements to the usability of the interface would also need to be implemented; all of the problems that caused negative codes in PEOU would be fairly addressed once they have been observed. The next technical improvement would be to set up automation for the creation of database templates to always keep these up-to-date, and prevent configuration drift to cause problems for the environments on creation.

Once improvements have been made to the system, more data can be gathered to make an assessment of whether or not the workflow is truly effective. There could be a full-scale pilot program at the case company to evaluate the adoption in a real-world context. For the cost difference in RQ4, we can move beyond estimations, instead conducting controlled experiments where different developers solve identical defects in pre-merge and post-merge contexts.

In conclusion, EPEs show clear potential in making customer-specific pre-merge validation a default activity rather than a late-stage exception in shared staging. In this thesis, we demonstrate that an EPE-enabled workflow can increase confidence and reduce defect-resolution effort by shifting defect discovery earlier in the delivery flow. However, for the workflow to have sustained value, focus must extend beyond tooling to ownership of data, infrastructure, and process. When these conditions are met, EPEs can serve as a practical foundation for earlier and more reliable validation.

Bibliography

[1] P. Lopes, P. Accioly, P. Borba, and V. Menezes, “Choosing the Right Git Workflow: A Comparative Analysis of Trunk-based vs. Branch-based Approaches.” [Online]. Available: https://arxiv.org/abs/2507.08943

[2] Edward Thomson, “Release Flow: How We Do Branching on the VSTS Team.” Accessed: 2018. [Online]. Available: https://devblogs.microsoft.com/devops/release-flow-how-we-do-branching-on-the-vsts-team/

[3] The GitHub Authors, “GitHub flow.” Accessed: 2025. [Online]. Available: https://docs.github.com/en/get-started/using-github/github-flow

[4] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[5] N. Forsgren, J. Humble, and G. Kim, *Accelerate: Building and scaling high performing technology organizations*. IT Revolution Press, 2018.

[6] F. Rahman and A. E. Hassan, “Environment Configuration Drift: Causes, Impacts, and Remedies,” *IEEE Software*, vol. 35, no. 6, pp. 63–69, 2018, doi: 10.1109/MS.2018.290100359.

[7] R. Shrestha and A. A. Nur Ali, “Configuration Management in Kubernetes Environments: A GitOps Approach,” in *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*, 2024, pp. 497–502. doi: 10.1109/UCC63386.2024.00077.

[8] T. K. Authors, “Homepage.” [Online]. Available: https://kubernetes.io/

[9] D. Team, “Goodbye Staging? Ephemeral Preview Environments and Database Branching Are Redefining QA and Release Workflows.” [Online]. Available: https://debugg.ai/resources/goodbye-staging-ephemeral-preview-environments-database-branching-qa-release-workflows

[10] F. Davis and F. Davis, “Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology,” *MIS Quarterly*, vol. 13, p. 319–, 1989, doi: 10.2307/249008.

[11] Netlify, “Deploy Previews.” 2024.

[12] GitLab, “Review Apps.” 2024.

[13] Vercel, “Preview Deployments.” 2024.

[14] X. Qu, "Configuration aware prioritization techniques in regression testing," in *2009 31st International Conference on Software Engineering - Companion Volume*, 2009, pp. 375–378. doi: 10.1109/ICSE-COMPANION.2009.5071025.

[15] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 426–437. doi: 10.1145/2970276.2970358.

[16] B. Vasilescu, Y. Yu, H. Wang, and P. Devanbu, "Continuous Integration and Software Quality: A Causal Explanatory Study," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ACM, 2015, pp. 805–816. doi: 10.1145/2786805.2786850.

[17] C. Yilmaz, M. B. Cohen, and A. M. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 20–34, 2005, doi: 10.1109/TSE.2005.140.

[18] C. Luo, S. Lyu, W. Wu, H. Zhang, D. Chu, and C. Hu, "Towards High-Strength Combinatorial Interaction Testing for Highly Configurable Software Systems," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1579–1591. doi: 10.1109/ICSE55347.2025.00113.

[19] H. Srikanth, M. B. Cohen, and X. Qu, "Reducing Field Failures in System Configurable Software: Cost-Based Prioritization," in *2009 20th International Symposium on Software Reliability Engineering*, 2009, pp. 61–70. doi: 10.1109/ISSRE.2009.26.

[20] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, 2007, pp. 129–139. doi: 10.1145/1273463.1273483.

[21] E. Soares, G. Sizilio, J. Santos, D. Costa, and U. Kulesza, "The Effects of Continuous Integration on Software Development: a Systematic Literature Review," p. , 2021, doi: 10.48550/arXiv.2103.05451.

[22] T. Dybå, D. I. Sjøberg, and N. B. Moe, "Investigating the Impact of Continuous Integration Practices on Software Development Outcomes," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 798–808. doi: 10.1109/ICSE.2018.00084.

[23] T. B. Authors, "Backstage." [Online]. Available: https://backstage.io/

[24] T. G. Authors, "Deploying with GitHub Actions." [Online]. Available: https://docs.github.com/en/actions/how-tos/deploy/configure-and-manage-deployments/control-deployments

[25] A. C. Authors, "Argo CD - Generating Applications with ApplicationSet." [Online]. Available: https://argo-cd.readthedocs.io/en/stable/operator-manual/applicationset/

[26] T. K. Authors, "Custom Resources." [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[27] A. Zerouali, R. Opdebeeck, and C. De Roover, "Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 523–533. doi: 10.1109/MSR59073.2023.00078.

[28] A. Authors, "ArgoCD - Overview." [Online]. Available: https://argo-cd.readthedocs.io/en/stable/

[29] J. Dobies and J. Wood, *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, 2020.

[30] A. C. Authors, "Argo CD - Git Generator." [Online]. Available: https://argo-cd.readthedocs.io/en/stable/operator-manual/applicationset/Generators-Git/#git-generator-files

[31] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009, doi: 10.1007/s10664-008-9102-8.

[32] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, no. 6, pp. 728–738, 1984, doi: 10.1109/TSE.1984.5010301.

[33] D. G. Malcolm, J. H. Roseboom, C. E. Clark, and W. Fazar, "Application of a technique for research and development program evaluation," *Operations Research*, vol. 7, no. 5, pp. 646–669, 1959, doi: 10.1287/opre.7.5.646.

[34] "ISO 9241-11:2018 Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts," no. ISO9241–11:2018. Geneva, Switzerland, 2018.

[35] A. Donker and P. Markopoulos, "A Comparison of Think-aloud, Questionnaires and Interviews for Testing Usability with Children," *Proceedings HCI 2002*, p. , 2002, doi: 10.1007/978-1-4471-0105-5_18.

[36] W. H. DeLone and E. R. McLean, "Information Systems Success: The Quest for the Dependent Variable," *Information Systems Research*, vol. 3, no. 1, pp. 60–95, 1992, doi: 10.1287/isre.3.1.60.

[37] S. McConnell, *Software Estimation: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2026-01-16

EXAMENSARBETE Early defect detection through ephemeral preview environments
Tidig defektdetektering med temporära förhandsvisningsmiljöer
STUDENTER Björn Tenje Persson, Fredrik Wastring
HANDLEDARE Ulf Asklund (LTH)
EXAMINATOR Sule Tekkesinoglu (LTH)

Early Defect Detection Through Ephemeral Preview Environments

POPULÄRVETENSKAPLIG SAMMANFATTNING **Björn Tenje Persson, Fredrik Wastring**

In software development bugs are often discovered too late in the process. This means extra work, for developers and testers. To find bugs earlier and shorten the process, this thesis suggests a new test workflow based on temporary copies of the entire system.

Rather than making one large release at long intervals, companies aim to speed up the release process by making continuous and smaller releases. This shift puts higher requirements on the time and quality of the testing, which puts a heavier load on the developers and quality assurance team. Despite increased testing, many bugs are missed due to time pressure and the sheer complexity of the software.

The thesis was conducted at a company that delivers highly configurable software. Depending on each customer and their configuration, the system can behave very differently across deployments. The root of the problem arises here, as in a best case scenario all of the configurations would need to be tested. Since that would take a lot of time and be very costly, local environments and shared test environments are used. However, neither would do a perfect job of resembling a production system and the risk of bugs leaking through would increase as time goes on.

To address these challenges, a web portal was introduced, where developers and testers create fully functional and isolated copies of the system on demand. Through the portal, they choose which code branch, version, and customer-like dataset the temporary environment should use. The remaining steps are automated, with the proper files retrieved and the environment set up within minutes of starting the process. This allows them to test changes in realistic conditions before moving on to the next stage.

The results of the thesis show that using a workflow based on temporary environments significantly reduces the time required to prepare test setups. By allowing the developers and testers to test in realistic conditions, they were able to find defects earlier. At this particular company, it has the potential to substantially reduce the rework time if the defects are caught and resolved during development, rather than at later stages.