Cloud Programming Languages and Infrastructure From Code: An Empirical Study

Georg Simhandl
Uwe Zdun
georg.simhandl@univie.ac.at
uwe.zdun@unvie.ac.at
Research Group Software Architecture, Faculty of Computer Science, University of Vienna
Vienna, Austria

Abstract

Infrastructure-from-Code (IfC) is a new approach to DevOps and an advancement of Infrastructure-as-Code (IaC). One of its key concepts is to provide a higher level of abstraction facilitated by new programming languages or software development kits, which automatically generate the necessary code and configurations to provision the infrastructure, deploy the application, and manage the cloud services. IfC approaches promise higher developer productivity by reducing DevOps-specific tasks and the expert knowledge required. However, empirical studies on developers' performance, perceived ease of use, and usability related to IfC are missing. We conducted a controlled experiment (n=40) to assess the usability of the cloud programming languages (PL) and software development kits (SDK). Both approaches involve similar effectiveness. We found that the PL-based approach was moderately less efficient but increased correctness with time spent on programming. Tracing generated infrastructure configurations from code was more challenging with the SDK-based approach. Applying thematic analysis, 19 themes emerged related to usability barriers, supporting factors, security, cloud cost, and enhancement areas. We conclude with five findings and future directions.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **General and reference → Empirical studies**; • **Software and its engineering → Domain specific languages**.

Keywords: Programming Language, Cloud, Infrastructure From Code, Empirical Study, Experiment

1 Introduction

Infrastructure-from-Code [5] defines approaches generating infrastructure and application deployment configurations from code. IfC combines software development and development operations (DevOps) [27] by abstracting the infrastructure-relevant declarations, e.g., infrastructure-as-code (IaC) [16]. The two primary approaches are IfC Programming Languages (PL), compiling to IaC and optimized application code accessing the generated IaC, and IfC Software Development Kits (SDK), offering language-agnostic programmatic interfaces to infrastructure resources and attributes, such as policies.

Whereas IaC deployment necessitates specialized knowledge in cloud architecture and subsequent code or model abstraction, IfC aims to simplify this process. It generates cloud infrastructure and deployment scripts from application source code, eliminating the need for a domain-specific language and reducing the risk of manual operations errors [5]. IfC thus offers a promising solution to streamlining the configuration of cloud resources and its provisioning process, using comprehensible abstractions and underlying transpilation processors.

A key challenge in IaC is security, particularly identity and access management, and ensuring the least privilege of resources. IfC aims to provide easy-to-use software language concepts to ensure these critical security objectives. Considering the potential of IfC, it is necessary to evaluate its usability, especially for novices. Our primary research question is: *What are the developer productivity (efficacy) and usability of the two different IfC concepts, i.e., SDK-based and PL-based approaches?* From this, we derive several sub-questions:

RQ₁ How do novices comprehend IfC language concepts of SDK- and PL-based approaches?

RQ₂ How efficient and effective are novices working with the two different approaches?

RQ₃ How do novices comprehend generated IaC artifacts and their link to SDK- and PL-based abstractions?

RQ₄ What are the two approaches' perceived usability and ease of use, good practices to expand, and critical barriers to adopting the IfC approaches from a novices' point of view?

We are particularly interested in finding differences in usability and developers' productivity and identifying underlying supporting factors and barriers affecting developer experience themes. Using the participant's feedback, we apply thematic analysis to identify critical success factors for cloud programming languages and aggregate the emerging themes into findings and future directions.

This paper is structured as follows: We describe the background and related work in Section 2, the experimental design adhering to guidelines for controlled experiments [14] in Section 3 and analyze the efficiency and effectiveness, measured by correctness and time to complete the task in Section 4. Themes that emerged from survey results are reported in Section 5. We further report the results and findings in Section 6, discuss threats to validity (Section 7), conclude with implications, and synthesize our findings with recommendations and potential future directions (Section 8).

2 Background and Related Work

Infrastructure as Code (IaC) has its roots in the mid-2000s, particularly with the release of Amazon's Web Services Elastic Compute Cloud (EC2) and declarative, domain-specific languages such as Puppet and Ansible. These early tools were soon followed by Terraform, which aimed to standardize cloud provisioning practices.

Initial IaC solutions like Ansible [3] and Chef [8] employed imperative scripting approaches. In contrast, declarative solutions such as Puppet [19] allowed developers to specify desired states, significantly enhancing adaptability and robustness. Modern IaC tools, including AWS CloudFormation [1], Terraform [24], and Pulumi [18], further this approach by utilizing typed, directed, acyclic resource graphs [26]. These tools also leverage general-purpose programming languages (GPL) like Go, Python, and TypeScript for defining infrastructure states.

The PIACERE project has contributed significantly to this field by developing the DevOps Modelling Language (DOML). This language describes cloud applications independently of specific cloud providers and IaC tools through a multi-layer approach, encompassing application, abstract, and concrete infrastructure layers. DOML enables developers to map software components to infrastructure elements, facilitating various deployment options [9]. A study by [23] involving 73 IT professionals revealed that, in practice, manual coordination is often employed for correct deployments despite

expectations of superior software delivery and operations performance through fully automated approaches.

A recent study identified 14 key cloud infrastructure practices and evaluated the limitations of current cloud automation technologies like Infrastructure as Code (IaC), noting their complexity and need for specialized knowledge. The authors introduced the *Infrastructure from Python Code* approach to address these issues, automatically generating cloud deployment templates from application source code. They evaluated it through a case study with 24 student development teams [4].

2.1 Infrastructure from Code Approaches

The first and second generations of IaC used domain-specific languages, e.g., Chef, Puppet, and Ansible, to configure infrastructure, leading to repetitive and verbose configurations. The third generation of IaC, e.g., Pulumi or CDK, uses GPLs to solve this problem. With the growing number of cloud resources, development and operation teams, infrastructure configurations, and runtime code are further separated. IfC tries to close this DevOps gap. Instead of separate infrastructure and application code, they eliminate the former, leaving only the application code, and the infrastructure is completely derived from code.

SDK-based IfC approaches represent the next step in evolution, mainly using an SDK to access cloud resources' functionalities and configurations. SDK-based approaches or approaches using a combination of SDKs and annotations rely on well-known GPLs, e.g., Python, Go, and TypeScript. For instance, Encore [13], Shuttle [22], Ampt [2], and Nitric [17] are characterized by GPLs and SDK to abstract infrastructure configurations.

Conversely, the PL-based approach takes a different path, introducing a new programming language and distinct execution concepts. For example, Darklang [12] and Wing [25] introduce new syntax and new concepts like *biphasic programming* [20], where the new programming language is used to express computations executed in two different phases, e.g., infrastructure configuration and application while remaining consistent behavior across the phases. Wing represents one of the first cloud programming languages to apply this concept to cloud-native development.

To compare the understandability of an SDK-based vs. a PL-based approach, we thoroughly evaluated IfC approaches according to availability, cloud-agnostic support, and production readiness criteria. We chose Nitric [17] and Wing [25]. Wing and Nitric aim to facilitate cloud application development by abstracting infrastructure concerns. However, their concepts differ significantly. Nitric is well suited as a baseline (control) as it uses established programming language concepts for infrastructure and application code, while Wing is a new programming language introducing new concepts.

WingLang (Wing) is a novel programming language for cloud development. It integrates cloud concepts directly

into application code. Wing transpiles applications into Terraform and JavaScript for cloud deployment.

Nitric provides Software Development Kits (SDKs) for existing languages, e.g., TypeScript, JavaScript, Python and Go. SDKs interact with a deployment engine and providers to deploy applications across various clouds. Both approaches, Nitric and Wing, aim to enhance productivity, but their implementation methods cater to different developer preferences and use cases.

Wing, as a programming language, introduces two major concepts. First, **Inflight** code implements application behavior, e.g., handling API requests and processing queue messages. It can be executed on various cloud computing platforms, including function services, e.g., AWS Lambda, containers, e.g., ECS, Kubernetes, and virtual machines or physical servers. Second, **Preflight** code runs once at compile time and transpiles this code to infrastructure configurations, e.g., Terraform. Preflight code has no special annotation and is Wing's default execution phase.

3 Experiment

3.1 Experimental Design

The user study was carefully designed, adhering to ethical and privacy-related requirements of the authors' institution and country's laws, i.e., participants were informed about the study's purpose, data collection procedure, participants' rights, and anonymization strategy.

The dependent variables, time to complete a task and correctness of the task, influenced by the independent variable (treatment groups), solution approach W (presenting the PL-based approach), and N (representing the SDK-based approach), are moderated by (confounding) factors, such as knowledge, skills, and work experience in the area of cloud computing, programming language proficiency and general software engineering related problem-solving skills. We assessed these factors with a pre-experiment survey.

3.2 Procedure

We recruited participants from the authors' institution and announced the experiment during a distributed systems engineering course two weeks before the event. Along with the announcement, we provided a reading list featuring expert articles on IaC and tutorials on SDK-based approaches (e.g., Pulumi and Nitric) and PL-based approaches (e.g., WingLang). The experiment was conducted in the institution's computer laboratory, where each workstation was equipped with Linux, pre-installed Nitric and Wing frameworks, and Visual Studio Code with Wing and TypeScript extensions.

To guide participants through the experiment, we utilized the Moodle learning platform. Participants were required to fill out a consent form with a unique pseudonym (e.g., B42) to ensure anonymity and minimize bias. Each participant entered this pseudonym at the start of the experiment.

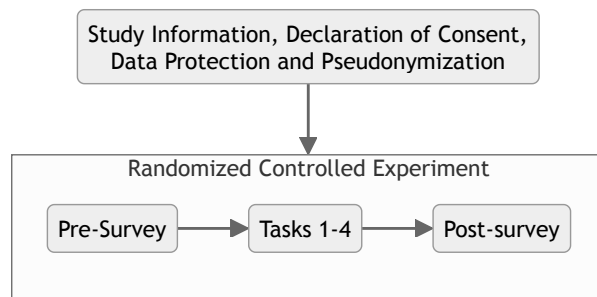Figure 1 shows the procedure We collected basic informa-

Figure 1. Experimental design and procedure of the study.

tion such as demographics, prior knowledge, and preferred programming languages. Tasks and materials were prepared for two experiment groups. Participants were instructed to record their start time for each task and, upon completion, to note the end time and rate the task's difficulty level and their confidence in their answers.

The final section of the survey assessed the perceived usefulness and ease of use of the programming languages. It also included open-ended questions regarding interesting and distracting features. In the post-hoc survey, participants were asked to rank the activities based on the time required to complete each task. For example, participants might spend most of their time reading the source code, consulting API documentation, writing source code, writing tests, and executing tests.

3.3 Material

To assess the understandability of key concepts of IfC in terms of correctness and time to complete the tasks, we prepared a URL shortener application implementation, both for Wing and Nitric. The simplified implementation and potential solution code excerpts can be found in Appendix A. The sample application uses three cloud services: The API Gateway, a simple key-value storage (a *bucket* in Wing), and an encrypted secret storage. The secret storage is used to store the API-Key to avoid API-Key leakage. The *G*ET-request handler retrieves the target URL from a key-value store, where the key is the *a*lias or short name, making a short URL, and relocates to the target URL. We provided a basic authentication middleware using the encrypted cloud storage for the API key to create new short to long URL mappings, realized by an POST-request handler. Both approaches transform or generate infrastructure code. To compare the source-code-to-configuration-code-tracability, we had to implement a new provider (plugin) for Nitric. The provider generates the necessary Terraform configuration in JSON format, similar to Wings transpilation output.

3.4 Tasks - Group A

To avoid potential sequencing effects, i.e., favoring the first tasks, and fatigue effects, i.e., decreasing correctness or increasing time to complete a task due to increasing cognitive load, we designed a controlled two-group intra-subject experiment consisting of Groups A and B.

Task 1 - Wing's Concept Comprehension: Participants were asked to read the code and comprehend the core concepts of Wing. We asked participants to check out a git repository of a simple *U*RL shortener cloud app, *c*ompile or *r*un the program, view the local cloud emulator, and count the number of cloud services used in the application. Using a multiple-choice test, we further examined participants' knowledge of the IfC approach. **Task 2 - Implementing using Wing**: Treatment *W* (Wing) represents the programming language approach that is similar to *T*ypeScript language and introduces Wing's *i*nflight and *p*reflight concept. We evaluate the understandability of these new language concepts by asking participants to read the code and extend the application. Participants were asked to write a new serverless (*Lambda*) function requiring understanding and applying both concepts. We prepared a *URLShortenerAlgorithm* in Javascript code. The goal of this task is to 1) reference an external source code, the *URLShortenerAlgorithm*, and 2) offload the URL shortening (hashing) to a serverless function. **Task 3 - Traceability of Nitric IfC**: This third task requires inspecting the infrastructure configuration and assessing the participant's understandability of the (compiled) output and the time to inspect and summarize it. Group A inspected the Terraform configuration generated by Nitric. **Task 4 - Implementation using Nitric**: To evaluate the understandability of the SDK-based approach, we asked participants to write a *DELETE*-request handler, requiring the participant to understand the concept of implied access control of the reference to a cloud service, here the key-value store, and adding the *delete* parameter in the *allow*-method. This task evaluates the cloud programming language's API and its documentation.

3.5 Tasks - Group B

Group B participants performed tasks involving equal effort but in a different order and adapted task descriptions. **Task 1 - Nitric Concept Comprehension**: This task corresponds to Task 1 for Group A, i.e., Wing's Concept Comprehension. We adapted the multiple choice questions to Nitric's core concept of *implied permissions*. **Task 2 - Implementing with Nitric**: This task corresponds to Task 4 of Group A (Implementing using Nitric). **Task 3 - Traceability of Wing IfC**: This task corresponds to Task 3 of Group A, i.e., Traceability of Nitric IfC. We slightly adapted the multiple-choice questions for Wing. **Task 4 - Implementing using Wing**: This task corresponds to Task 2 of Group A (Implementing using Wing).

3.6 Participants

The participants are students in their final year of a Bachelor's or the beginning of their Master's program with a major in computer science, data science, or business informatics. Participants were granted course credits as an incentive. In Figure 2, we provide an overview of participants' demographics. We invited 42 participants.

One participant opted out and did not sign the consent sheet, and another participant did not finish the experiment, mentioning in the post-hoc survey that "without any experience (it is) impossible to finish the task." Hence, this subject was excluded from further analysis.
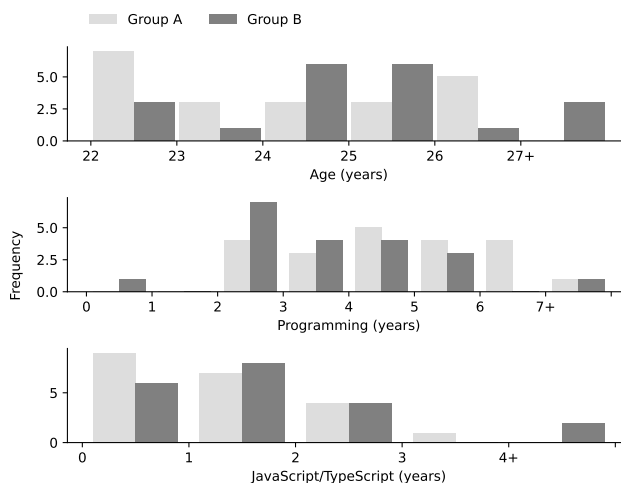

Figure 2. Demographics of participants.

The remaining 40 participants were evenly divided into two groups. Group A consisted of three females and 17 males, with eight in a Master's program and 12 in a Bachelor's program, including five in Business Informatics, two in Data Science, and 13 in Computer Science. Group B also had three females and 17 males, with ten in a Master's program and ten in a Bachelor's program, including 15 in Computer Science and five in Business Informatics.

Participants in Group A were slightly younger, with 23.7 (±1.89), than in Group B, with 25.35 (±3.87) years. We provide the complete dataset, results, and necessary material to replicate the study[1].

4 Analysis

We aim to analyze the impact of two IfC concepts on developer effectiveness and efficiency: the PL-based approach, Wing (*W*), and the SDK-based approach, Nitric (*N*). Given the indications of non-normality in the metric-dependent variables of correctness and response time, parametric testing assumptions are violated, rendering parametric tests

---

[1] https://doi.org/10.5281/zenodo.12622490

unsuitable. The non-parametric Kruskal-Wallis test cannot be utilized instead, as it assumes that distribution shapes do not differ except in their central locations. Thus, due to the properties of our data, we use Cliff's delta [10], a robust non-parametric measure that remains unaffected by distribution and non-normal data changes. It is recommended as a robust approach in empirical software engineering [15].

Furthermore, we analyzed responses to open-ended questions using thematic analysis. This involved systematically coding the data, translating the codes into potential themes, reviewing these themes by examining and augmenting the codes with quotes, and minimizing the overlap between themes. To understand participants' mental models, we employed the card-sorting technique. The resulting themes are defined and reported with descriptive names, and the trustworthiness of the synthesis is thoroughly assessed.

4.1 Outlier detection

For the 41 participants, we aggregated the results by $\delta_{C_s} = C_{s(W)} - C_{s(N)}$, and $\delta_{T_s} = T_{s(W)} - T_{s(N)}$ for each subject to detect outliers, e.g., when a subject (s) significantly used less time to complete the programming tasks, using a one-tailed t-test ($\alpha = 0.005$). In group A, the mean δ_{T_s} is 33 minutes (± 18 minutes and 10 seconds). One group A subject spent significantly less, and three spent no time on the second programming task. In group B, the mean δ_{T_s} is 15 minutes and 15 seconds (± 15 minutes and 17 seconds). Three participants of group B spent significantly less, and two participants spent no time on the second programming task. These participants were excluded from the analysis for Task 4.

4.2 Hypotheses

We designed the experiment to compare Treatment Wing, W, and Control, Nitric, N in terms of effectiveness, measured by correctness C, and efficiency, measured by time to complete the task T, of performing realistic comprehension and programming tasks. Group B in Task 1 summarizes code N (Nitric), Task 2 corresponds to Task 4 in Group A, Task 3 reads code and inspects output in W, and Task 4 corresponds to Task 2 in Group A.

For each of the concept comprehension, the programming and the code-to-infrastructure-traceability tasks and RQ_1 to RQ_3 respectively, the following null (H_O) and alternative (H_{Alt}) hypotheses can be stated:

$\mathbf{H}_{0_C}$ Treatment W does not affect the number of correctly answered questions or decrease it compared to N.

$\mathbf{H}_{0_T}$ Treatment W does not affect the average time needed to answer a question correctly or increase it compared to N.

$\mathbf{H}_{Alt_C}$ Treatment W increases the number of correctly answered questions compared to N.

$\mathbf{H}_{Alt_T}$ Treatment W decreases the average time needed to answer a question compared to N correctly.

For RQ_4 we formulate the null hypothesis, that Treatment W does not affect the average usability and ease of use or decrease them compared to N $\mathbf{H}_{0_U}$ and the related alternative hypothesis $\mathbf{H}_{Alt_U}$ whereas Treatment W increases the usability and ease of use compared to N.

4.3 Analysis Results

Detailed results are in Table 4.4 summarizing each task's result mean and standard deviation and the hypothesis testing results using Cliff's δ, including the related effect size description. Specifically, the table shows the results for Task 1, including sub-tasks for code summarization (Resource Count), and the multiple-choice test (Concept Comprehension, Task 2 (Programming Task), and Task 3, including sub-tasks for locating configuration and comprehending infrastructure configurations.

4.4 Concept Comprehension

For $\mathbf{RQ}_1$, we evaluated the understandability of PL-based and SDK-based approaches with two sub-tasks: The correct number of declared cloud resources, C_{1_1}, was evaluated binarily, and participants gained one point if they identified the three resources. Each correct answer on the multiple-choice test was awarded 0.2 points, and incorrect answers were deducted 0.2 points, with a minimum score of 0 and a maximum of 1 point. Figure 3 illustrates the cumulative results of both sub-tasks with a kernel density plot.

In Group A, which worked on the PL-based approach, participants struggled with two statements: Only 57% of the participants said the following statement is incorrect: *The* GET*-request handler, handling redirections, needs to be compiled to infrastructure configurations to get executed on cloud computing resources.* Only 62% said that *The function* create *cannot be executed during preflight because inflight APIs assume all resources have already been deployed.* is correct.

In Group B, which worked on the SDK-based approach, four out of five multiple-choice questions were mostly answered correctly. The most difficult-to-answer question was about Nitric's *implied permission concept.* 60% of participants incorrectly assumed that *during deployment the CLI command converts accessor decorators, e.g.,* @allow("read","write"), *of resources to policies based on the targeted cloud provider.*

Comparing the correctness of both sub-tasks (C_{1_1} and C_{1_2}), we cannot reject H_0C_1. However, Cliff's $\delta > 0.3$ indicates a medium effect size for the duration (H_0T_1), with the PL-based approach (Wing) decreasing the average time needed to answer correctly compared to the SDK-based approach (Nitric). The results for the perceived difficulty of Task 1 ($\delta = -0.36$) suggest that the PL-based approach was moderately easier to understand. There was a negligible difference in the confidence that the answers were correct.

Task	Sub-Task	Wing Mean	Std	Nitric Mean	Std	Cliffs δ	Effect size
Task 1	C_{1_1} Correctness of Resource Count	0.450	0.510	0.350	0.489	0.100	negligible
	C_{1_2} Correctness of Concept Comprehension	0.530	0.339	0.600	0.311	-0.115	negligible
	T_1 Total time spent on Task 1	15.300	4.231	18.900	6.069	-0.378	medium
	Perceived Task's Difficulty	2.950	0.945	3.600	0.681	-0.360	medium
	Perceived Confidence	3.350	1.089	3.600	1.046	-0.113	negligible
Task 2	C_2 Correctness of Programming Task	0.375	0.393	0.375	0.275	-0.037	negligible
	T_2 Total time spent on Task 2	35.850	15.776	26.400	11.905	0.380	medium
	Perceived Task's Difficulty	4.250	1.020	4.400	0.883	-0.087	negligible
	Perceived Confidence	4.000	1.376	4.350	0.933	-0.085	negligible
Task 3	C_{3_1} Correctness of Code-to-Infrastructure Traceability	0.533	0.516	0.333	0.488	0.200	small
	C_{3_2} Correctness of Configuration Comprehension	0.533	0.516	0.467	0.516	0.067	negligible
	T_3 Total time spent on Task 3	10.133	7.453	9.467	3.907	-0.098	negligible
	Perceived Task's Difficulty	3.267	1.280	4.400	0.910	-0.498	large
	Perceived Confidence	3.333	1.345	4.400	0.910	-0.471	medium

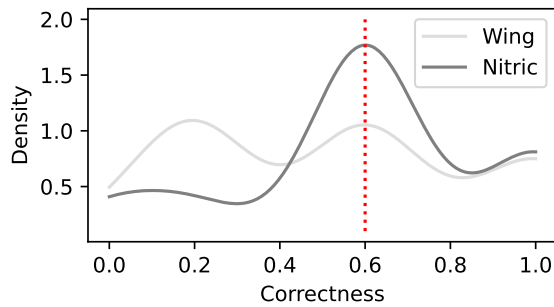**Table 1.** Evaluation results for Tasks 1 to 3 for Treatment Wing and Control Nitric.

Figure 3. Understandability of concepts Wing vs. Nitric

4.5 Programming Task and Developer Productivity

Developer productivity (efficacy) is measured by the time to complete a programming task (T_2) and the correctness of the implementation (C_2). Groups A and B were tasked with implementing a Serverless function using WingLang and extending a Nitric-based API with a request handler to delete alias-to-URL mappings.

For Wing, correctness (C_2) was evaluated on a three-level scale: fully correct (1 point), partially correct (0.5 points, e.g., implementing the lambda function but failing to invoke it correctly), and incorrect (0 points). For Nitric, the evaluation considered two sub-tasks: fully correct (1 point) submissions included correctly set IAM permissions by adding the delete parameter, correctly implementing the request handler, and identifying the resulting configuration changes (KeyValueStoreDelete) in the generated Terraform configurations file.

Among 20 participants in each group, 14 from Group B (Nitric) and only 5 from Group A (Wing) correctly set the permissions. Most participants (95%) working with Nitric could not trace the required changes in the IaC file.

Figure 4 shows a scatter plot of C_2 (correctness) and T_2 (completion time) for both Wing (Treatment) and Nitric (Control). The trend line (indicated by the red line) shows that for Wing, correctness increases with time spent, while Nitric shows no evidence of a learning effect. Comparing the correctness (C_2) of the submitted solutions, we cannot reject $H_0 C_2$, as both approaches yield similar correctness. Participants spent more time ($T2$) completing the task with Wing than with Nitric, resulting in a medium effect size. Therefore, we cannot reject the null hypothesis $H_{0_{T_2}}$; Treatment W (Wing) increases the average time needed to complete the task compared to N (Nitric).

Participants working with both treatments perceived the task *difficult to very difficult* and were *slightly to very uncertain* about the correctness of their submission. There is a *negligible* difference when comparing the confidence that the answers were correct.

4.6 Cloud Resource Traceability

Wing and Nitric automatically generate infrastructure from code but differ significantly in their approaches (see Section 2.1). To address **RQ**$_3$, we evaluated traceability, i.e., the ability to link code to generated infrastructure configurations, by comparing the PL- and SDK-based approaches.

Participants were tasked with localizing generated IAM configurations in the Terraform code and understanding specific configuration parameters, such as the retention period for AWS CloudWatch logs (Wing) or identifying assets configuring IAM roles for buckets (Nitric).
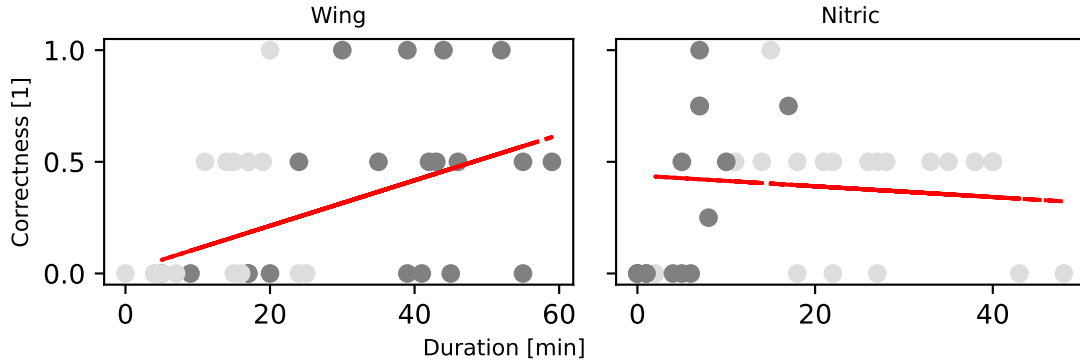
Figure 4. Scatter plot of correctness, C_2 of the implementation versus time spent for the programming task, T_2, for Wing and Nitric (combined groups). The red line illustrates the apparent trend.
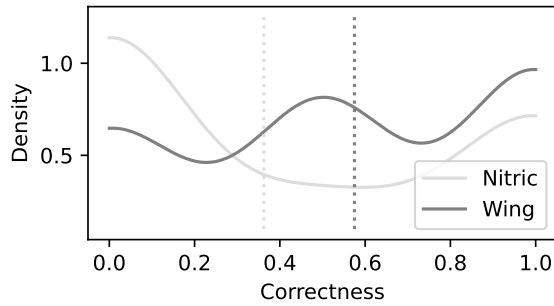

Figure 5. Kernel density plot of the Correctness, C_3 of Task 3 on code to infrastructure configuration traceability with Wing and Nitric

Figure 5 illustrates the aggregated results of both subtasks, C_{2_1} and C_{2_2}, using a kernel density plot. Participants using Wing (Treatment) achieved slightly higher correctness (C_{3_1}) than those using Nitric (Control), with a small effect size ($\delta = 0.2$). Wing moderately increased the number of correctly answered questions, providing weak support for the alternative hypothesis $H_{Alt}C_3$.

However, as indicated by Cliff's $\delta > 0.067$, the negligible effect size for the duration (H_0T_3) suggests that Wing does not significantly affect the average time needed to answer a question compared to Nitric correctly. The results for perceived difficulty ($\delta = -0.498$) show that the PL-based approach was significantly easier to understand. Additionally, the medium effect size indicates that participants using Wing were moderately more confident in the correctness of their answers.

4.7 Usability and Ease-of-use

Regarding $\mathbf{RQ}_4$, we assessed the two approaches' perceived usefulness and ease of use. Participants rated Wing's usefulness 2.725 (± 0.816) as *useful to neutral* and perceived Nitric slightly less useful, 3.275 (± 0.8). The perceived ease of use

of the PL-based approach was an average of 2.775 (± 1.062) *easy to neutral*, while participants rated Nitric an average of 3.45 (± 1.01) *neutral to difficult*. The effect size for usefulness ($\delta = -0.0244$) and ease of use ($\delta = -0.0844$) is in both cases negligible. The quantitative analysis can only partially answer $\mathbf{RQ}_4$ related to the perceived usability and ease of use of both approaches.

Qualitative analysis is needed to evaluate users' feedback on supporting factors and critical barriers to adopting cloud programming approaches, facilitates good practices, and analyze enhancement areas to expand IfC approaches in the future.

5 Thematic Analyis

Thematic Analysis is a robust qualitative research method utilized to identify and elicit themes from qualitative discursive data [7] also frequently used in software engineering research [11]. We use this method to understand participants' experiences with the different IfC approaches. We systematically coded the data from the semi-structured survey answers with 178 initial codes and identified 21 recurring themes. We organized these themes into five categories: Usability challenges and barriers and supporting factors affecting the developer experience, security, cost estimation, and suggested areas of enhancement from a novices' point of view. Figure 6 illustrates the shares of the individual themes and how they relate to the participant's experience with cloud computing. We used a Sankey chart, a type of flow diagram in which the width of the arrows is proportional to the flow rate. It is used to visualize the flow of data from one set of values to another, helping to identify patterns and relationships in the data [21]. Participants with *advanced* knowledge and skills worked more than three years with cloud development toolkits or IaC and at least two years classified as *intermediate*. The cluster of participants at the *beginner*-level is defined by at least one year of experience with cloud computing while *Novices* didn't gain any experience yet.

5.1 Usability Challenges and Barriers

Several usability challenges and barriers hinder developer experience. In this context, seven themes emerged from the participant's survey.

Learning Curve: This theme reflects the challenges of learning and adapting to new abstractions, including new programming paradigms, novel frameworks, or new languages. For example, B18 highlighted this theme, expressing that the PL-based approach is more difficult to learn, and B9, working with this PL-based approach, mentions the complexity and that *(...) it takes time to get (yourself) around. The code looks bulk, especially JSON*. Similarly, subject A25 meant that the *Paradigm can take some time to click with newcomers*. To a lesser extent, this theme can be found in the context of SDK-based approaches. B2, for example, mentions that: *it is not very easy to get used to it*.

Language Syntax Barriers: The learning curve is closely related to the degree of difficulty learning a new programming language or framework. To this regard, A4 mentioned the *new functions and new methods which again have to be incorporated* in the context of the SDK-based approach. Subject B4, also working with this SDK-based approach, said *I didn't feel like I understood any concept deeply, a lot of magic*. Novel languages similar to known language designs also lead to confusion or even unacceptance if there is a lack of knowledge of the basic concepts. Participant A9 mentioned that *the syntax is kind of confusing to me, don't really know why since it's close to JavaScript, but still, I had trouble with it.* while A22 meant: *I did not like that it was based on JavaScript. I do not know JavaScript and therefore struggled with everything (...)*. In contrast, SDK-based approaches like Nitric strive to support many popular languages. Missing language support, documentation, and examples may limit the developer's experience. In this context, participant A10 perceived the approach as *not language agnostic*.

Usability Constraints: This theme encompasses the various aspects of code and project structure that contribute to difficulties in understanding, navigating, and managing the software effectively. It highlights issues like confusing code structure, excessive configurations, and overwhelming file organization that can hinder developers' productivity and comprehension. 15 participants mentioned usability constraints. With the PL-based approach, participant A14 noticed *less information of user errors*, while B21 had trouble navigating the code structure and meant *it was hard to get used to the order (...)*. In this regard, participant A14 mentioned the *rather verbose* code structure. In contrast, the SDK-based approach has *limited support for some advanced features* according to A11. The perceived usability was also constrained by setting up and performing *testing*, e.g., B3 mentioned the unsuccessful attempts to test the app and infrastructure generated with the SDK-based approach.

Performance Lags: Another critical factor affecting the usability of new cloud programming languages or frameworks is performance. Participants A11 and A24 reported *occasional performance lags* in the context of the PL-based approach.

Infrastructure Configuration Complexity: This theme captures developers' challenges with understanding, locating, and managing numerous intricate configurations, particularly in Terraform JSON format. Infrastructure configurations of both approaches were perceived as equally difficult to comprehend. The *generated terraform files are difficult to understand* (B4). While novices mention that they *don't understand where to find the Terraform JSON file*, users with intermediate experience levels find that it is *hard to keep (the) overview* with *many different folders* (A17), containing the Terraform templates. B8 mentions that *there are quite a few configuration files I don't like too much.*

Insufficient support resources: This theme encompasses the difficulties developers face due to the lack of community support, outdated or inadequate documentation, and overly simplistic examples and tutorials that do not meet their needs. Participant A23 summarizes these problems: The *docu is partially outdated, extremely lacking in examples, only JS supported (at least with the given example project), onboarding/getting started tutorials are not nearly sufficient for the given task. Yes, this is the same feedback as for the PL-based approach. They suffer from the same problem*. A24 also mentions that the *(...) documentation can be insufficient for complex scenarios*. Subject B16 emphasizes the importance of support resources: *The main problem is that it is hard to find help with the (...) SDK-based approach if you cannot solve your problem with the documentation alone. The limited documentation and community support can make troubleshooting difficult (...)* as participant B7 mentioned.

Unclear use cases: Finally, this theme captures developers' challenges when unsure about the specific use cases or scenarios where the approach or the provided features should be applied. This is emphasized by participant B8: *I'm not quite sure which use case the PL-based approach(sic!) even tries to solve.*

5.2 Supporting Factors

Novel programming languages and frameworks, in particular, require accessible onboarding, including comprehensive documentation, tutorials, and working examples to support the learning process. Easy-to-understand APIs and intuitive usage are equally important factors in adopting a new approach. Specific to *C*loud Programming Languages and Frameworks is the seamless integration with cloud service providers and tooling support, for instance, enhanced debugging and visualization.

Accessible Onboarding: This theme highlights the availability of good tutorials, comprehensive documentation, and easy setup processes (32 mentions). Participants emphasized
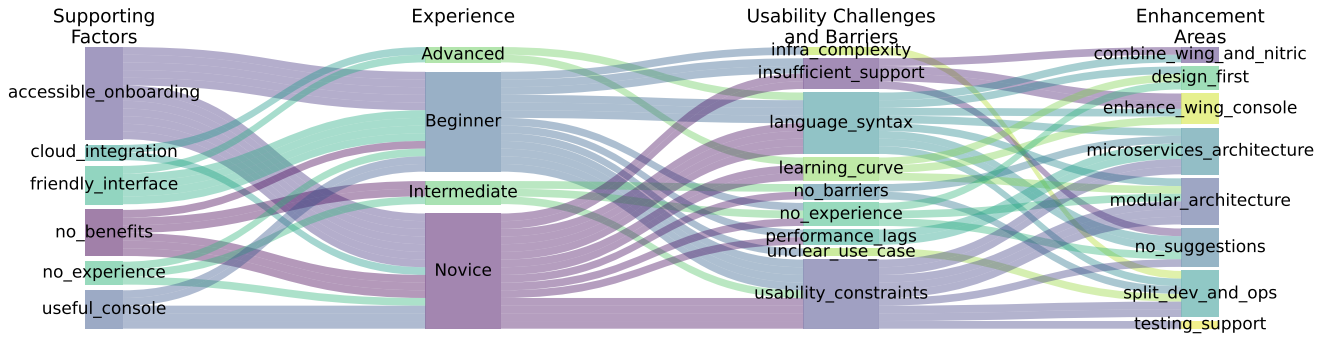
Figure 6. Sankey Chart of themes emerged from the survey

that the SDK-based approach can be used with programming languages I already knew (A21), and is *easy to understand* (A11). The PL-based approach is *easy to use once understanding the basics provided a comprehensive documentation* (A14). *Tutorials help to start from zero and get some basic output.* (B29).

User-Friendly Interface: This theme reflects the importance of intuitive usage, easy-to-understand APIs, straightforward creation of infrastructure and cloud resources, minimal coding, and clean interfaces. Participant B10, for example, perceives *the management of cloud resources is abstracted* useful and mentions *I don't have to think about that (...) and can switch from Azure to AWS and back without changing (the) code*. After completing the tasks using the PL-based approach, participant A3 meant that *Wing is very uncomplicated and straightforward if you know JavaScript*. At the same time, A12 said *that you have to write considerably less code*.

Seamless Cloud Integration: The advantages of cloud agnosticism, simplified identity and access management (IAM), worry-free infrastructure configurations, cloud security, and *seamless integration with cloud services and ease of deployment* (A24) were mentioned by eleven participants. B4 commented *that I do not have to worry about the infrastructure*. Like the SDK-based approach, subject A3 working with Wing mentioned: *You can implement cloud function very easily*.

Enhanced Debugging and Visualization: This theme captures the benefits of having local simulation and visualization tools that aid in debugging, tracking changes, receiving immediate feedback, and visualizing component interactions. Both experiment treatments provide a local cloud development dashboard. Participant A4 perceived Wing's *Console useful, as it (...) keeps track of your created functions*, and A9 highlighted that *it is intuitive to use* while A12 found that the local simulation *makes it easy to track and test your resources and find any problems in your code*. The SDK-based approach also visualizes component interactions and testing tools, whereas participant A9 highlighted the intuitive

user interface providing *all the details needed*. In total, eight participants found the local emulation environment useful.

5.3 Security Concerns

In the context of the programming task using the SDK-based and PL-based approaches, we asked participants to assess potential vulnerabilities, threats and exploits when deploying the application with a cloud service provider. One of the most mentioned themes is *Risks of Data Breaches*, involving many aspects such as *API Key Leakage*. Further, two themes emerged regarding security concerns: *Cloud Service Provider Dependency*, and *Uncertain Cloud Costs*.

Risks of Data Breaches and Attacks: This theme encompasses the various security threats that can lead to data breaches, including API key leakage, injection attacks, and account hijacking. The survey results reveal the participant's awareness of various identified threats, risks, and vulnerabilities. Participant A3 commented that the application developed with PL- and SDK-based approaches *(...) is exposed to many threats, like someone intercepting the data while transferring.*, and B17, for instance mentioned that *potential threats are data breaches, DDoS, (...) data leakage* due to *misconfiguraitons* or *badly written code could leave some things exposed* (A22). Eight participants mentioned distributed denial-of-service attacks, e.g., participant A20 thinks *the application might be exposed to DDoS attacks*. A12's assessment also includes DDoS attacks, where *(...) attackers could create a huge load of service requests, resulting in high costs*. Participant B8 said that the *biggest risk is probably exploding costs*. This contributes to the recurring theme *Uncertain Cloud Costs* (detailed in 5.4).

A data breach is a security incident that involves the unauthorized release of private and sensitive information to the public. API key leakage due to misconfiguration or broken user authenticationrepresents one of the *(...) highest risks in cloud-native systems*, as B8 put it. In this regard, subject B4 expands this to potential risks when *(...) IAM and AWS accounts (are) compromised*. Serverless architectures are exposed to further risks, as discussed by Participant B15: *Data stored*
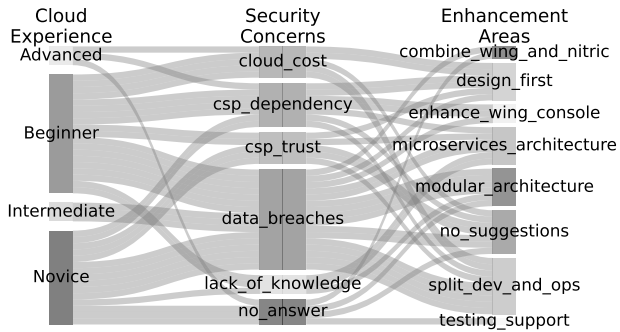
Figure 7. Sankey diagram of themes grouped by cloud experience, security concerns, and enhancement areas.

Figure 8. Participant's estimation of the URLShortener's daily cloud Costs for 1K, 10K, and 1M requests per hour (log scale).

in the Lambda execution context that persists across invocations can be exploited if not properly managed; also, API keys hard-coded in the application or stored in publicly accessible repositories can be exploited. Participant B16 augments this analysis: *Attempts to hack the password/gain unauthorized access or app-level exploits, as otherwise, the main target would be the cloud service provider themselves.* referring another recurring theme, *Dependency on Cloud Service Providers.*

Dependency on Cloud Service Providers: This theme reflects the security concerns and vulnerabilities related to relying on cloud service providers, such as API changes, and the reliance on providers for infrastructure and cloud security. In this context, participant's opinions diverge. Participant A9, for example, mentions: *In my opinion, I think it's generally safe to use these cloud service providers since they pride themselves on security and usability* and represents a group of participants thinking that security is the *cloud provider's responsibility* (B24) and *(...) trust the cloud service itself, there should be no problem* (A7). This contrasts with the other group of participants, who are more skeptical. *The Cloud Provider might change its API, pricing plan, cease to exist* (B29) summarizes the risks of reliance on cloud services providers. Figure 7 illustrates the proportion of themes discussed by participants with different experience levels and security concerns. It shows that participants with no cloud programming experience tend to trust cloud service providers more than participants with little experience, while a larger share of participants experienced in cloud computing rather address the risks of relying on cloud service providers concerning security.

5.4 Uncertain Cloud Costs

This theme captures the uncertainty regarding the costs of cloud resource usage. Participant B8 estimates that the *biggest risk is probably exploding costs.* Like B4 and A25, participant A12 addresses the risk of financial damage: *With DDoS attacks, attackers could create a huge load of service requests resulting in man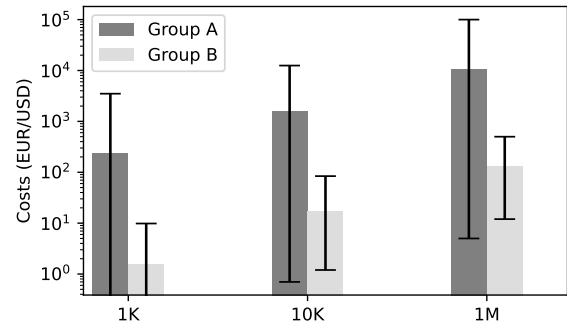y costs.* We asked participants to estimate the costs of the URL-shortening application they worked on during the experiment. Figure 5.4 shows the average and range of cloud costs per group. The estimates range from EUR 0.00-3500.00 for thousand (1K), EUR 0.70-12,500.00 for ten thousand (10K), and EUR 5.00-EUR 100,000.00 for one million (1M) requests per hour for group A, and EUR 0.00-9.84 for 1K, EUR 1.20-84.00 for 10K, and EUR 12.00-500.00 per 1M requests per hour for group B.

5.5 Areas of Enhancement

In the context of a larger application, participants proposed the following enhancements:

Modular Architecture and Microservices Architectural Style: This theme highlights the importance of using a microservice architectural style, structuring code within single repositories, separating application and infrastructure code, and utilizing specific tools provided by cloud service providers. Six participants suggested applying a modular architecture to scale cloud-native applications, and five proposed following the principles of the microservices architectural style. A21 proposes to *(...) a microservices architecture where each component is a separate service with its own repository and visualize the big picture with a detailed architecture diagram showing the interactions between services, data flows, and dependencies.* Subject B19 would choose a modular architecture with separate *code repositories* and mentions applying *monitoring, logging, code review, (and) collaboration* best practices.

Comprehensive Testing Support: This theme focuses on the necessity of robust testing support, including unit, integration, local, and cloud testing. For example, subject A23 would suggest comprehensive *testing tools* to improve the developer experience and productivity. Participant B29 suggests supporting *test component by component* in a modular architecture.

Design First: This theme emphasizes the importance of starting with high-level models. In subject A12's opinion, the PL-based approach to the application is *already pretty well*

visualized with the Wing console. Before implementing every-thing, though, I would probably create some UML diagrams to design the architecture. A25 reverts to existing UML de-sign tools and would start *with architecture diagrams in lucid chart.* Similarly, B29 would *fully plan every single component and how they interact before starting with the implementation* and B11 would also use high-level architecture diagrams, particularly to *(...) show interactions between components.*

Some participants, e.g., B10, suggest integrating the *design-first*-approach in cloud programming languages using a *mod-ular way by using Wing because Wing simplifies the visual-ization of the project.*

Split Development and Operations: Six participants would revert to the separation of development and develop-ment operations (DevOps) practices. For instance, partici-pant A5 would *use something(sic!) like AWS or Google cloud. They have good predefined components.* Subject B8 *would use the tools of the specific cloud I am using.* and justifies the suggestions: *There are many tools to organize code and visu-alize from AWS or Azure..* Also, B4 *(...) would prefer to split infrastructure and code* and proposes: *I would like to visualize and manage my infrastructure through a GUI.* B16 proposed *different docker containers managed via an IaC approach.* Par-ticipant A13 *(...) would leave to people who understand how to do that* and B21 also suggests that *everything regarding cloud connection and stuff, is handled by its(sic!) own department.*

Enhance Cloud Programming Language: This theme captures the proposals to enhance the PL-based approach. Some participants perceive the cloud programming languages and frameworks already usable for larger applications, em-phasizing the cloud *simulation* and visualization of the com-ponents (B6). A23 would use the PL-based approach, *some form of easy-to-read and write/understand language, close to one already known to most users or supporting multiple users.* and in addition, the subject proposes a *graphical represen-tation of the created application with monitoring and testing tools as a bonus.* Participant A12 *(...) would give every resource in wing their own file or directory and (...) separate the code between preflight and inflight functions,* and further suggests to *(...) divide the code into different single responsible mod-ules (authentication, user management, data processing* and emphasizes to *(...) stay independent from AWS, Azure, etc. by using wing.*

6   Results and Findings

The combined quantitative and thematic analysis of the con-trolled experiment on developer experience of cloud pro-gramming languages reveals five major findings. Regard-ing the comprehensibility of concepts, participants work-ing with the PL-based approach were moderately faster in summarizing the application code and answering concept

comprehension-related questions than the SDK-based ap-proach. Despite the novelty of Wing's programming lan-guage features, the comprehension task was perceived as moderately easier. As participants had unrestricted access to the Internet, e.g., to search the API documentation, the completion time may also be affected by the efficiency of retrieving task-relevant information. Comprehensive and up-to-date documentation is crucial for new programming languages and frameworks. Further analysis reveals that easy setup and local cloud emulation support users with little or no cloud programming experience. In contrast, seamless cloud integration and easy-to-use command line interfaces are important for advanced users and users with working experience with cloud services.

Finding 1 - Insufficient Support Resources: The results suggest the need for up-to-date documentation, clear use cases, extensive working examples, and comprehensive tuto-rials tailored to different target groups for both approaches. Community support, such as public forums like StackOver-flow, is also crucial. High-level abstractions, e.g., new pro-gramming languages features, the *implied permission* con-cept, and complexity of code-to-infrastructure transpilation, may hinder comprehensibility, cause confusion, and reduce developer productivity. The *explicit-implicit trade-off* needs to be carefully designed in the context of software language engineering. The quantitative analysis of programming tasks (Task 2) shows no difference in effectiveness between PL- and SDK-based approaches, but the PL-based approach requires more time. Developers must learn and adopt new paradigms and abstractions, which can be confusing, especially given the similarity to TypeScript. As Figure 4 shows, there is a learning effect with the PL-based approach: more correct implementations are achieved with increasing time. In con-trast, the SDK-based approach lacks support for some users' favorite programming languages, which, along with high-level abstractions and concept comprehension issues, may explain the decreasing trend in developer productivity. Other usability challenges and barriers affect the efficacy, e.g., per-formance lags and missing support for advanced features for debugging and testing. Many participants failed to set up or extend existing test code, particularly with the SDK-based approach. These factors are critical for larger cloud-native applications.

Finding 2 - Modular Architecture for Larger Cloud Applications: Many participants suggested a modular archi-tecture and proposed applying the microservice architectural style and designing the application first to improve the code structure and maintainability. In addition, local cloud emu-lation environments and comprehensive testing and mon-itoring support can significantly improve developers' pro-ductivity. The complexity of infrastructure configurations, assessed with Task 3, represents a main barrier, particularly for beginners in cloud computing. The quantitative analysis clearly emphasizes the novices's difficulty in understanding

infrastructure configurations. The Terraform JSON format is difficult to read and trace back to invocations in code. Terraform templates organized in many folders, as used in the SDK-based approach, are overwhelming.

Finding 3 - Lack of Infrastructure Traceability and Testing Support: The traceability of code to generated infrastructure is crucial in managing infrastructure configurations. The study results suggest that configuration templates decrease comprehensibility. This is also reflected in the perceived task difficulty, which was significantly higher with Terraform templates than single file Terraform JSON format. The thematic analysis (Sec. 5) reveals two more findings:

Finding 4 - Cloud Cost: Facing the uncertainty of estimating the financial implications (see Section 5.3) of cloud-native applications, the integration of cost estimation tools, including graph-based estimation methods, to predict and manage cloud expenses to reduce the risk of *exploding costs* is evident. [6], for instance, proposes a directed graph-based cost model enabling monetary cost estimations from code through static analysis, integrated into a code editor for immediate visualization. This approach can simplify cost exploration, making it part of the development process without relying on external tools.

Finding 5 - Cloud Security Risks and Cloud Service Provider Dependency: Besides usability constraints, language and syntax comprehensibility, and uncertain cloud costs, cloud security risks and the dependency on cloud service providers are crucial barriers to adopting cloud programming approaches. While some novices and beginners tend to trust cloud service provider's security measures and would remain cloud service provider independent using a cloud programming framework, participants with advanced and intermediate cloud computing knowledge are skeptical about the dependency on cloud service providers. The emerging themes regarding security concerns (Section 5.3) related to enhancement areas suggest that these participants would mitigate these risks by splitting application and infrastructure code and using cloud provider-specific and comprehensive testing and observability tools.

7 Threats to Validity

Threats to Internal Validity: There were no disruptions during the experimental sessions. Participants received an introduction and could ask questions individually without affecting the overall process. The short session duration minimized maturation effects, which were not observed. Each participant contributed only once, eliminating inter-session learning effects. Different task scopes prevented intra-session learning from favoring either Treatment or Control. Equal scoring opportunities prevented instrumental bias, and random group assignment eliminated selection bias. Measures to minimize cross-contamination included restricting access to the survey and source code to session

times, spacing out sessions, and using only laboratory PCs with identical setups.

Threats to External Validity: The small sample size may affect statistical significance. To mitigate this, we used robust statistical methods suitable for our sample size, ensuring accurate analysis.

Threats to Construct Validity In Section 3, we examined correctness and time to complete tasks, common metrics for assessing understandability and efficacy. However, other metrics might better capture understandability. We supplemented our quantitative analysis with a thematic analysis of survey results. Solutions were excluded based on limitations like mandatory paid subscriptions or extensive training requirements, e.g., Shuttle's use of Rust. This may have led to a non-representative set of IfC approaches being evaluated. Nitric and WingLang were selected based on availability, cloud-agnostic support, and production readiness criteria.

Threats to Content Validity: No threats to content validity were identified. Tasks were relevant and unbiased, instructions were clear, and participant expertise was representative of the target population. Multiple data sources and controlled contextual factors ensured comprehensive task coverage and systematic thematic analysis.

Threats to Conclusion Validity: We mitigated threats to conclusion validity by providing the information material to all participants, offering the necessary prerequisite knowledge for their participation in the study. Any inconsistencies or ambiguities in the experiment material would have affected both groups equally, ensuring fairness.

8 Conclusion

The study identifies barriers and supporting factors in adopting two Infrastructure as Code (IfC) approaches, highlighting areas for improvement. While both approaches aimed to enhance productivity, no significant difference in concept comprehension or implementation correctness was observed. However, participants using the PL-based approach improved implementation correctness over time, suggesting that cloud-native languages, with comprehensive support, can improve code quality and maintainability.

Regardless of the approach, clear use cases and comprehensive support resources tailored to specific target groups are critical for success. Based on our findings, future research on programming languages for IfC should focus on enhancing local cloud emulation, expanding cloud integration support, and developing tools for code-to-infrastructure traceability, cost estimation, and systematic security checks to improve developer productivity and system maintainability.

9 Acknowledgments

A   Code Excerpts

```
bring cloud;

// Preflight code runs once, compiles to IaC
let mapping = new cloud.Bucket();
let api = new cloud.Api() as "shorturl";

api.get("/:alias", inflight (req) => {
    let alias = req.vars.get("alias");
    try {
        let target = mapping.get(alias);
        return {
            status: 307,
            headers: { location: target }
        };
    } catch e {
        return { status: 404, };
    }
});
```

Figure 9. Comparison of a simple short URL forwarding service: Wing version.

```
import * as nitric from "@nitric/sdk";

// Creates a readable/writable reference to a
// KeyValue store
const mapping = nitric.kv('Map').allow('get','set');
const api = nitric.api("main");

api.get("/:alias", async (ctx) => {
    const { alias }  = ctx.req.params;
    try {
        const kv = await mapping.get(alias);

        ctx.res.status = 307;
        ctx.res.headers['location'] = [kv.target];

    } catch (error) {
        ctx.res.status = 404;
    }
});
```

Figure 10. Comparison of a simple short URL forwarding service: Nitric version.

```
let hashFunc = new cloud.Function(
    inflight (s: str?): str => {
        return StringShortenAlgo.hash(s);
});

api.post("/alias", inflight (
  req: cloud.ApiRequest) => {
    ...
    let alias = hashFunc.invoke(url);
    mapping.put(alias,url);
});
```

Figure 11. Excerpt of a possible solution for the task *Implementing using Wing*.

```
const mapping =
  nitric.kv('Map').allow('get','set','delete');

api.delete("/:alias", async (ctx) => {
  try {
    const { alias } = ctx.req.params;
    await mapping.delete(alias);
    ctx.res.status = 204;
  } catch (error) {
    ctx.res.status = 400;
  }
});
```

Figure 12. Excerpt of a possible solution for the task *Implementing using Nitric*.

References

[1] Amazon Web Services, Inc. 2024. *Infrastructure As Code Provisioning Tool - AWS CloudFormation.* Amazon Web Services, Inc. Retrieved May 21, 2024 from https://aws.amazon.com/cloudformation/

[2] Ampt Web Services, Inc. 2024. *The most efficient way to get things done in the cloud.* Ampt Web Services, Inc. Retrieved June 21, 2024 from https://getampt.com

[3] Red Hat 2024. *Ansible Collaborative.* Red Hat. Retrieved May 21, 2024 from https://www.ansible.com

[4] Itzhak Aviv, Ruti Gafni, Sofia Sherman, Berta Aviv, Asher Sterkin, and Etzik Bega. 2023. Cloud infrastructure from python code–breaking the barriers of cloud deployment. In *European Conference on Software Architecture, ECSA.*

[5] Itzhak Aviv, Ruti Gafni, Sofia Sherman, Berta Aviv, Asher Sterkin, and Etzik Bega. 2023. Infrastructure From Code: The Next Generation of Cloud Lifecycle Automation. *IEEE Software* 40, 1 (2023), 42–49. https://doi.org/10.1109/MS.2022.3209958

[6] Lukas Böhme, Tom Beckmann, Sebastian Baltes, and Robert Hirschfeld. 2023. A Penny a Function: Towards Cost Transparent Cloud Programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments* (Cascais, Portugal) *(PAINT 2023).* Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3623504.3623566

[7] Virginia Braun and Victoria Clarke. 2012. *Thematic analysis.* American Psychological Association. 65–81 pages. https://doi.org/10.1037/13620-004

[8] Progress Software Corporation 2024. *Chef Software DevOps Automation Solutions.* Progress Software Corporation. Retrieved May 21, 2024 from https://www.chef.io

[9] Michele Chiari, Elisabetta Di Nitto, Adrián Noguero Mucientes, and Bin Xiang. 2022. Developing a New DevOps Modelling Language to Support the Creation of Infrastructure as Code. In *European Conference on Service-Oriented and Cloud Computing.* Springer, 88–93. https://doi.org/10.1007/978-3-031-23298-5_8

[10] Norman Cliff. 2010. Answering Ordinal Questions with Ordinal Data Using Ordinal Statistics. *Multivariate Behavioral Research* 31 (06 2010), 331–350. https://doi.org/10.1207/s15327906mbr3103_4

[11] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement.* IEEE, 275–284. https://doi.org/10.1109/ESEM.2011.36

[12] Dark Inc. 2024. *darklang.* Dark Inc. Retrieved June 27, 2024 from https://darklang.com

[13] Encoretivity AB 2024. *Development Platform for type-safe distributed systems.* Encoretivity AB. Retrieved June 24, 2024 from https://encore.dev

[14] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. 2008. Reporting experiments in software engineering. *Guide to advanced empirical software engineering* (2008), 201–228. https://doi.org/10.1007/978-1-84800-044-5_8

[15] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust statistical methods for empirical software engineering. *Empirical Software Engineering* 22 (2017), 579–630. https://doi.org/10.1007/s10664-016-9437-5

[16] Kief Morris. 2020. *Infrastructure as code.* O'Reilly Media.

[17] Nitric Inc. 2024. *Next Generation Cloud Development.* Nitric Inc. Retrieved June 27, 2024 from https://nitric.io

[18] Pulumi Corporation 2024. *Pulumi - Infrastructure as Code in Any Programming Language.* Pulumi Corporation. Retrieved May 20, 2024 from https://www.pulumi.com

[19] Perforce Software, Inc. 2024. *Puppet Infrastructure and IT Automation at Scale.* Perforce Software, Inc. Retrieved May 21, 2024 from https://www.puppet.com

[20] Chris Rybicki. 2024. *Exploring biphasic programming: a new approach in language design.* Retrieved June 30, 2024 from https://rybicki.io/blog/2024/06/30/biphasic-programming.html

[21] Mario Schmidt. 2008. The Sankey diagram in energy and material flow management: part II: methodology and current applications. *Journal of industrial ecology* 12, 2 (2008), 173–185. https://doi.org/10.1111/j.1530-9290.2008.00015.x

[22] Openquery Ltd. 2024. *Build Backends Fast.* Openquery Ltd. Retrieved June 21, 2024 from https://www.shuttle.rs

[23] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. Automating serverless deployments for DevOps organizations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 57–69. https://doi.org/10.1145/3468264.3468575

[24] HashiCorp 2024. *Terraform.* HashiCorp. Retrieved May 20, 2024 from https://www.terraform.io

[25] Wing Cloud, Inc. 2024. *A programming language for the cloud.* Wing Cloud, Inc. Retrieved June 27, 2024 from https://www.winglang.io

[26] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. 2020. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-Physical Systems* 35 (2020), 63–75. https://doi.org/10.1007/s00450-019-00412-x

[27] Liming Zhu, Len Bass, and George Champlin-Scharff. 2016. DevOps and its practices. *IEEE software* 33, 3 (2016), 32–34. https://doi.org/10.1109/MS.2016.81