# Performance Evaluation of Intent-Based Networking Scenarios: A GitOps and Nephio Approach

Saptarshi Ghosh, Ioannis Mavromatis, Konstantinos Antonakoglou, and Konstantinos Katsaros
Digital Catapult, United Kingdom
Emails: {saptarshi.ghosh, ioannis.mavromatis, konstantinos.antonakoglou, kostas.katsaros}@digicatapult.org.uk

***Abstract*—GitOps has emerged as a foundational paradigm for managing cloud-native infrastructures by enabling declarative configuration, version-controlled state, and automated reconciliation between intents and runtime deployments. Despite its widespread adoption, the performance and scalability of GitOps tools in Intent-Based Networking (IBN) scenarios are insufficiently evaluated. This paper presents a reproducible, metric-driven benchmarking, assessing the latency and resource overheads of three widely used GitOps operators: Argo CD, Flux CD, and ConfigSync. We conduct controlled experiments under both single- and multi-intent scenarios, capturing key performance indicators such as latency and resource consumption. Our results highlight trade-offs between the tools in terms of determinism, resource efficiency, and responsiveness. We further investigate a realistic orchestration scenario, using Nephio as our orchestrator, to quantify the processing latency and overhead in declarative end-to-end deployment pipelines. Our findings can offer valuable insights for tool selection and optimisation in future autonomous network orchestration systems.**

***Index Terms*—GitOps, Intent-Based Networking, Nephio, Benchmarking, Cloud-Native Orchestration**

## I. Introduction

THE increasing complexity of modern cloud-native, Intent-Based Networking (IBN) [1] infrastructure has stimulated a shift from imperative to declarative deployment models. GitOps [2] has emerged as a compelling operational paradigm that uses Git as the Single Source of Truth (SSoT) for managing declarative infrastructure and application configurations. By continually synchronising the runtime states of deployments in Kubernetes (K8s) clusters with their desired states stored in the associated Git repositories through automated agents (i.e., Reconciliation Operators), GitOps enhances auditability, traceability, and overall system reliability.

In the context of cloud-native IBN, network orchestrators automate the deployment of desired network states, including all necessary assets, such as Virtual Network Functions (VNFs) and their interconnections, from a declarative manifest. For this paper, we use Nephio [3] as our desired orchestrator. Nephio captures an "Intent" as configuration values defined through Kubernetes Resource Model (KRM), decoupled from the application code and declarative deployment manifests, a concept referred to as Configuration as Data (CaD) [3]. GitOps provides an automated Continuous Deployment (CD) pipeline by tracking the desired states in a Git repository, comparing them with the corresponding runtime states of the orchestrated VNFs, and synchronising the states as soon as it detects any drift between them. When GitOps and CaD are combined, we have an IBN orchestrator capable of translating customer intent into a set of desired states, encoded in declarative manifests, and pushing it to the Git repository for deployment. To that extent, in CaD-enabled frameworks like Nephio, the terms "intent" & "desired-state" become synonymous. In such a framework, all Key Performance Indicators (KPIs) (e.g., reconciler's latency, resource utilisation, etc.) contribute to the End-to-End (E2E) performance of an IBN Orchestration [4], [5].

This paper builds upon the above principles and presents a systematic benchmarking of a cloud-native IBN orchestrator, broken down into two parts, i.e., the GitOps and the CaD pipelines. We first design a reproducible benchmarking pipeline using Kubernetes, GitLab [6] and three leading GitOps tools (ArgoCD [7], FluxCD [8], and ConfigSync [9]) to simulate realistic GitOps workflows. Later, we evaluate a CaD pipeline based on Nephio. We capture a set of latency and resource utilisation metrics, and provide an empirical evaluation under various scaling conditions. Our work aims to offer a quantitative performance baseline informing practitioners and researchers about the strengths and limitations of GitOps-enabled IBN orchestrators.

The remainder of this paper is organised as follows: Sec. II outlines related works, Sec. III describes the system design of the benchmarking system and its integration with Nephio. Sec. IV presents the experimental results and their analysis, and summarises the findings. We conclude in Sec. V summarising the contribution and future outlook of this work.

## II. State-of-the-Art & Motivation

This work builds upon our previous work [10], that demonstrates a bespoke IBN Orchestration platform, named Cloud-native Autonomous Management and Intent-based Orchestrator (CAMINO), utilising Nephio and Kubernetes, where we experienced certain limitations of ConfigSync. In this section, we first summarise the state of the art in GitOps and CaD, as well as their utilisation in cloud-native use cases, followed by a gap analysis that motivates this study.

The authors in [11] recommend using GitOps over DevOps, demonstrating the advantages of a pull-based, declarative deployment approach over its push-based counterpart through configuration changes and rollbacks using Argo CD. The study [12] presents a comprehensive guide to implementing Flux CD with a Kubernetes cluster. However, it lacks any quantitative results. The work [13] demonstrates a proof-of-concept using tools included in the Cloud Native Computing Foundation (CNCF) landscape, which validates the feasibility of GitOps in an IoT Edge-computing solution utilising Argo CD. However, it could not provide any remarks regarding scaling due to the limited number of worker nodes in the cluster. The analysis in [14] compares the computational overhead between GitOps and DevOps, indicating a trade-off between efficiency and cost. It shows a 17% reduction in total lines of code, accompanied by a 53% decrease in the average number of lines of code per file. However, there is a 75% increase in the number of files, along with an 80% increase in the depth of the directory tree. The methodology proposed in [15] demonstrates an Argo-CD-based Containerised Network Function (CNF) configuration framework using YANG Models and NETCONF, with scenarios for full and partial configuration changes. The implementation described in [16] is closest to our work, which utilises Nephio for IBN on a KIND-based K8s cluster on OpenStack and deploys a fixed Private 5G topology by integrating the Argo CD operator, replacing ConfigSync. Although it provides quantitative results on resource utilisation, it lacks in measuring latency. Finally, the evaluation in [17] compares the performance of reconciliation with that of traditional configuration management using Argo CD and Ansible [18], respectively. The performance analysis reveals that Argo CD consistently outperforms Ansible in handling misconfiguration changes, network configuration drift, and dependency updates.

Based on the ground study above, we identify two research gaps that we intend to address. First, the lack of a quantitative performance comparison based on latency and resource utilisation of the GitOps tools; and second, a comparison between single or simultaneous intent deployments (i.e., state changes), investigating how the reconciliation operators perform under different scaling conditions. The primary motivation for this work is to enhance CAMINO's performance by introducing a more capable reconciler. Building a bespoke experimental pipeline, we intend to evaluate the different frameworks in detail, identify an alternative to ConfigSync as the reconciler for CAMINO, and share our findings with the research community for further considerations.

## III. System Design

This section outlines the high-level system design of the benchmarking methodology, highlighting the algorithms employed and integration with the Nephio platform. Table I lists the various notations used in the following text.

### A. Benchmarking System

Fig. 1 illustrates the schematic diagram of our GitOps benchmarking pipeline. It comprises a benchmarking system, a Git repository serving as the SSoT, and the Kubernetes (K8s) cluster coexisting with three GitOps operators: Argo CD, Flux CD, and ConfigSync. Table II summarises the testbed specification. The control plane of each GitOps tool is isolated within its respective namespace, which hosts the reconciliation

TABLE I: List of Notations Used

| Notation | Description |
|---|---|
| $EP$ | Experiment Parameters $\{p, m, r, c\}$ |
| $p \in$ [a-z]$^+$ | Prefix String to identify GitOps tool |
| $m \in \mathbb{N}$ | Maximum number of deployments or replicas |
| $r \in \mathbb{N}$ | Number of representation |
| $c \in \mathbb{N}$ | Step count or range $[1 : m : c]$ |
| $S_{\text{template}}$ | Template of Desired State Manifest: YAML File |
| $R_{\text{template}}$ | Template of Reconciler Manifest: YAML File |
| $S_{\text{desired}}$ | Desired State of a deployment: YAML File |
| $S_{\text{runtime}}$ | Runtime State of a deployment: YAML File |
| $O_{\text{recon}}$ | Reconciler Operator Instance |
| $\Delta S$ | State Drift: $\texttt{Diff}(S_{\text{desired}}, S_{\text{runtime}})$ |
| $t_{\text{push}} \in \mathbb{Q}$ | Time to Push (in Sec.) |
| $t_{\text{recon}} \in \mathbb{Q}$ | Time to Reconcile (in Sec.) |
| $t_{\text{deploy}} \in \mathbb{Q}$ | Time to Deploy (in Sec.) |
| $t_{\text{healthy}} \in \mathbb{Q}$ | Time to reach Healthy status (in Sec.) |
| $t_{\text{hydrate}} \in \mathbb{Q}$ | Time to Hydrate a Dry Kpt package (in Sec.) |
| $t_{\text{inproc}} \in \mathbb{Q}$ | Nephio's Intent Processing Time |
| $u_{\text{cpu}} \in \mathbb{Q}$ | CPU Utilization by active reconciler in Millicore |
| $u_{\text{mem}} \in \mathbb{Q}$ | Memory Utilization by active reconciler in MiB |
| $K_{\text{attr}}$ | KPI Attributes |
| $K_{\text{attr}}^{(m,r,c)}$ | Aggregated KPI Attributes per experiment |
| $[l : u : c]$ | Discrete range: $l$ to $u$ (inclusive) with increment $c$ |

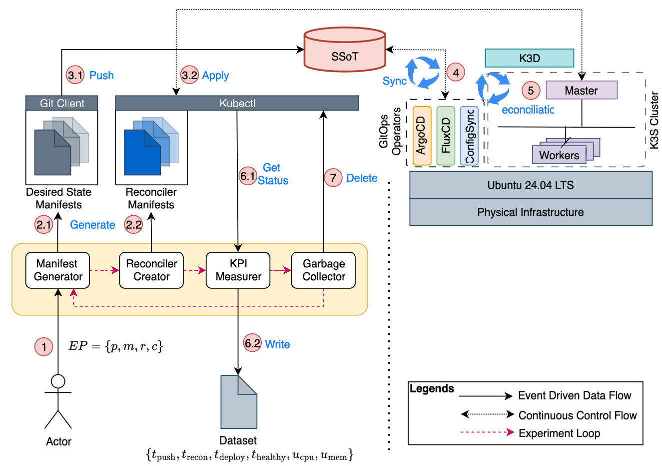


Fig. 1: Benchmarking pipeline for our GitOps investigation

operators and other control plane components. We measure the resource consumption of these namespaces to compare the GitOps tools in isolation. The lifecycle of an intent is divided into four stages (Generation, Synchronisation, Reconciliation, and Deployment stages), while the benchmarking experiments loop in four phases (Manifest Generation, Reconciler Generation, KPI Measurement, and Garbage Collection).

The Actor initiates an experiment through the Manifest Generation stage by providing a set of parameters $EP = \{p, m, r, c\}$, where $p$ denotes a string (prefix) to select the GitOps tools to use and name related variables, $m$ is the maximum number of application manifest (apps) or replicas to deploy, $r$ represents the number of repetitions of the same experiment, and $c$ is the step count of the range $[1 : m]$. The benchmarking system runs as a four-phase experimentation loop. In Phase 1, the *Manifest Generator* generates a set of manifest files based on $EP$ from a state template $S_{\text{template}}$ representing a collection of Desired States $\{S_{\text{desired}}\}_{i=1}^{n}$ where $n$ is the number of deployments (multi-app) or replicas (single-

TABLE II: Testbed Specifications

| Resource | Specification |
|---|---|
| CPU | 2 Sockets, 4 Cores each, clocked at 4.2 GHz |
| RAM | 16 GB DDR4 |
| Kubernetes Cluster | K3s with K3d wrapper |

app). Therefore, the experimentation loop is set to run for $|\{S_{\text{desired}}\}| = \lceil \frac{mr}{c} \rceil$ iterations (Alg. 1). Phase 1 concludes by invoking a Git Push operation that uploads the desired states into the Git repository (SSoT) with a specific directory structure. We measure the time to push as $t_{\text{push}}$. In Phase 2, the *Reconciler Generator* generates a set of reconciler object manifests $\{O_{\text{recon}}\}_{i=1}^{n}$ corresponding to $S^i_{\text{desired}}$ derived from a reconciler template $R^{(p)}_{\text{template}}$, depending on the GitOps tools identified by the prefix $p$ (Alg. 2).

In the implementation, instead of creating separate repositories for each GitOps operator to hook, we create a single repository containing multiple directories, each representing an $S^i_{\text{desired}}$ and specified their unique directory path in the corresponding $O^i_{\text{recon}}$ with a shared Git URL. Applying the manifests $\{O^i_{\text{recon}} | i \in [1 : m : c]\}$ establishes Webhooks dynamically between the $S^i_{\text{desired}}$ at SSoT and the $O^i_{\text{recon}}$ operators to track them. The GitOps operators compare the state drift $\Delta S$ as the drift between the tracked desired states $S^i_{\text{desired}}$ and their corresponding runtime states $S^i_{\text{runtime}}$. Although realising $\Delta S$ numerically is challenging, for convenience we shall denote $\Delta S = 0$ if $S^i_{\text{desired}} = S^i_{\text{runtime}} \forall i$ and $\Delta S \neq 0$ otherwise. In our implementation, we determine $\Delta S$ by comparing the revision string of the repository. We establish a Webhook between the SSoT and the GitOps operators authenticated by a Personal Access Token (PAT) to keep the desired state in sync. In Stages 2 and 3 of the workflow, the GitOps operator synchronises the $\{S_{\text{desired}}\}_{i=1}^{n}$ as the associated Webhook detects a Push operation, followed by calculating $\Delta S$. We measure the synchronisation and reconciliation time taken by the operators as $t_{\text{sync}}$ and $t_{\text{recon}}$, respectively.

In Stage 4 of the E2E workflow, Kubernetes manages deployments of the desired states with a drift. We measure the time to deploy as $t_{\text{deploy}}$ and the time it takes for deployment to become healthy as $t_{\text{healthy}}$. In Phase 3 of the experiment loop, our *Results Collector* fetches the respective namespaces of the GitOps operators, various status messages and metrics (CPU and Memory consumption) from the Kubernetes master, namely, $K_{\text{attr}} = \{t_{\text{push}}, t_{\text{sync}}, t_{\text{recon}}, t_{\text{deploy}}, t_{\text{healthy}}, u_{\text{cpu}}, u_{\text{mem}}\}$ (Alg. 3). Finally, after measurements collection, the benchmarking system releases all the resources by clearing all associated deployments, K8s CRDs and namespaces before looping back to Phase 1.

The remainder of this section outlines each of the benchmarking phases. We performed two types of experiments for each GitOps tool: first, a single-app deployment with multiple replicas, and second, a multiple-app deployment with single replicas each, where the single-app benchmarking measures the GitOps tools' reaction time to $\Delta S$, the multi-app benchmarking measures the ability to handle simultaneous $\Delta S$ with resource consumption.

**Algorithm 1:** Generate Kubernetes manifests

```
Input: n, p, d_target
Output: {S^1_desired, ..., S^n_desired}
for i ← 1 to n do
    name ← "{p}-app-{i}"                          // app name
    ns ← "{p}-ns-{i}"                             // namespace
    label ← "{p}-label-{i}"                       // label
    makeDir("{d_target}/{name}")
    S^i_desired ← S_template(name, ns, label)     // Generate S^i_desired
                                                  manifest using HELM Template S_template
    store(S^i_desired)                            // write to manifest
return {S^i_desired}^n_{i=1}
```

**Algorithm 2:** Generate reconciler manifests

```
Input: n, p
Output: {O^1_recon, ..., O^n_recon}
for i = 1 to n do
    rec ← "{p}-rec-{i}"                           // reconciler name
    ns ← ""{p}-ns-{i}"                            // namespace
    url ← args[cluster-url]                       // K8S cluster
    br ← args[git-branch]                         // git branch to watch
    rdir ← args[repo-dir]                         // sub-dir of the br
    if p ∈ {argo, csync, flux} then
        O^i_recon ← R^(p)_template (rec, ns, url, br, rdir)
            // Generate O^i_recon manifest using HELM Template R^(p)_template
        store(O^i_recon)                          // store manifest file
return {O^i_recon}^n_{i=1}
```

*1) Phase 1: Manifest Generation:* The manifest generation algorithm (Alg. 1) takes three inputs $(n, d_{\text{target}}, p) \,|\, \forall n \in [1 : m : c]$, and $d_{target}$ is the name of the sub-directory(s) Alg. 1 creates to segregate application manifests. We use Helm to dynamically generate manifests with the app name, target namespace $NS$, and label derived from $p$.

*2) Phase 2: Reconciler Generation:* The reconciler generator algorithm (Alg. 2) generates $\{O^i_{\text{recon}}\}_{i=1}^{n}$ of length $n = 1$ for single-app and $n$ for multi-app benchmarks. However, the manifest definition varies based on the GitOps operator to target. We use operator-specific reconciler templates $R^{(p)}_{\text{template}} \mid p \in \{\texttt{"argo"}, \texttt{"flux"}, \texttt{"csync"}\}$ and Helm Charts to generate $\{O^i_{\text{recon}}\}_{i=1}^{n}$ dynamically, linking with the appropriate Git repository with a PAT.

*3) Phase 3: KPI Measurement:* The performance-measuring algorithm (Alg. 3) is identical across all GitOps tools. We maintained this homogeneity to keep the comparison agnostic of the GitOps tools, despite some tools, e.g., ArgoCD, providing APIs to fetch the measurements. We leverage the K8s status logs fetched from Kubectl to extract attributes in $K_{\texttt{attr}}$. For an experiment defined by parameters $EP$, Alg. 3 measures the KPI attributes $K^{(m,r,c)}_{\texttt{attr}}$ as Eq. 1.

**Algorithm 3:** Measuring performance metrics

**Output:** $K_{\text{attr}}$

**foreach** $(app, ns) \in \{(app, ns)\}$ **do**
  $t_{\text{start}} \leftarrow$ `tstamp()` // timestamp
  git.push($S_{\text{desired}}$[app]) // push desired state to repo
  $t_{\text{push}} \leftarrow$ `tstamp()` $- t_{\text{start}}$ // calculate $t_{\text{push}}$
  $t_{\text{start}} \leftarrow$ `tstamp()` // timer reset
  **while** $\Delta S \neq 0$ **do**
    continue
  // drift: git revision update
  $t_{\text{sync}} \leftarrow$ `tstamp()` $- t_{\text{start}}$ // calculate $t_{\text{sync}}$
  $t_{\text{start}} \leftarrow$ `tstamp()` // timer reset
  **while** *true* **do**
    $t_{\text{create}} \leftarrow$ `app.CreationTimestamp`
    **if** $t_{create}(ns) \neq$ `Null` **then**
      **break**
  $t_{\text{deploy}} \leftarrow t_{\text{create}} - t_{\text{start}}$ // calculate $t_{\text{deploy}}$
  $t_{\text{start}} \leftarrow$ `tstamp()` // timer reset
  **while** *true* **do**
    $n_{\text{ready}} \leftarrow$ `app.availableReplicas`
    $n_{\text{desired}} \leftarrow$ `app.replicas`
    **if** $n_{ready} = n_{desired}$ ***and*** $n_{desired} \neq n_{desired}$ **then**
      **break**
  // wait until all deployed states becomes healthy
  $t_{\text{healthy}} \leftarrow$ `tstamp()` $- t_{\text{start}}$
  $u_{\text{cpu}}, u_{\text{mem}} \leftarrow \texttt{Top}(O^{(p)}_{\text{recon}}, ns)$ // Measure resource utilization using Top tool
$K_{\text{attr}} \leftarrow \{t_{\text{push}}, t_{\text{sync}}, t_{\text{deploy}}, t_{\text{healthy}}, u_{\text{cpu}}, u_{\text{mem}}\}$
**return** $K_{attr}$

$$K^{(m,r,c)}_{\text{attr}} = \Big\{ \frac{1}{r} \sum_{j=1}^{r} \Big( \sum_{i=1}^{k} t_{\text{push}}, \sum_{i=1}^{k} t_{\text{sync}}, \sum_{i=1}^{k} t_{\text{recon}}, \sum_{i=1}^{k} t_{\text{deploy}}, \sum_{i=1}^{k} t_{\text{healthy}}, \frac{1}{k} \Big( \sum_{i=1}^{k} u_{\text{cpu}}, \sum_{i=1}^{k} u_{\text{mem}} \Big) \Big) \Big\} \mid \forall k \in [1:m:c] \tag{1}$$

We measure the trend of the attributes in $K^{(m,r,c)}_{\text{attr}}$ with respect to $m$ to identify a correlation between them.

*4) Phase 4: Garbage Collection:* After every iteration of the experiment loop, the garbage collection phase runs, communicating with the K8s cluster through Kubectl to release all occupied resources by deleting any associated K8s objects (e.g., CRDs, Deployments, and Namespaces). This process eliminates the risk of miscalculation due to progressive loading by resetting the cluster after each iteration.

Our four-phase experiment loop iterates through the three different tools, considering all possible combinations of replicates and applications, and repeats $r$ times for each experiment to ensure the statistical validity of our results. Finally, all results are saved in a CSV file for further data analysis.

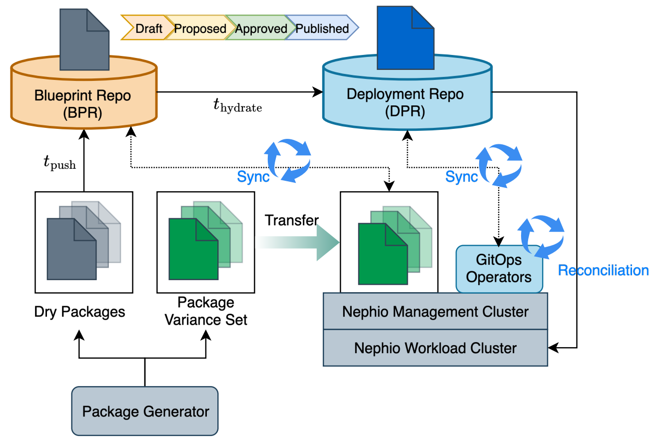


Fig. 2: Nephio Integration with Benchmarking System

### *B. Nephio Integration*

Nephio's intent processing mechanism involves two repositories, namely Blueprint (BPR) and Deployment (DPR). BPR holds the intents, i.e., the set of $S_{\text{desired}}$ as Kpt packages (also called *Dry Packages*). A Package Variant Set (PVS) tracks the revisions of Dry Packages at the BPR through Webhooks. Nephio leverages Package Orchestrator (Porch) to process Dry Packages ("hydrate" them), inserting configuration data into the templates and turning them into *Hydrated Packages*. During the Hydration Process, Nephio handles the Git operations, creating a branch named "Draft" in the DPR and merging it with the main branch after processing. During the Hydration process, a package migrates through four sequential states, i.e., draft, proposed, approved, and published. Finally, published packages reside in the DPR's main branch, which the operators from $O_{\text{recon}}$ track; therefore, the integration with the Benchmarking system sets the DPR as the SSoT, as described in the previous section, introducing an additional latency for Hydration ($t_{\text{hydrate}}$). Fig. 2 depicts the integration workflow.

## IV. Result & Analysis

This section covers the findings by analysing the experimental results of deploying a lightweight Nginx application template as $S_{\text{template}}$. We cover three test-case scenarios, first, single app deployment with scaling it by its replica with a range of $[1:100:10]$ with $r = 20$ and measuring $t_{\text{push}}$, $t_{\text{sync}}$, $t_{\text{recon}}$, $t_{\text{deploy}}$; second, simultaneous multiple app deployment with single replica with a range of $[1:90:10]$ with $r = 20$, measuring $t_{\text{recon}}, t_{\text{deploy}}, t_{\text{healthy}}, u_{\text{cpu}}, u_{\text{mem}}$; third the latency introduced by Nephio processing single and multiple Intents. For all the experiments, we have grouped the aggregated measurements ($K^{(m,r,c)}_{attr}$) by their respective reconciler prefix $p$ and traced their corresponding trajectory of median trend lines to establish our conclusion. For multi-app deployment, we

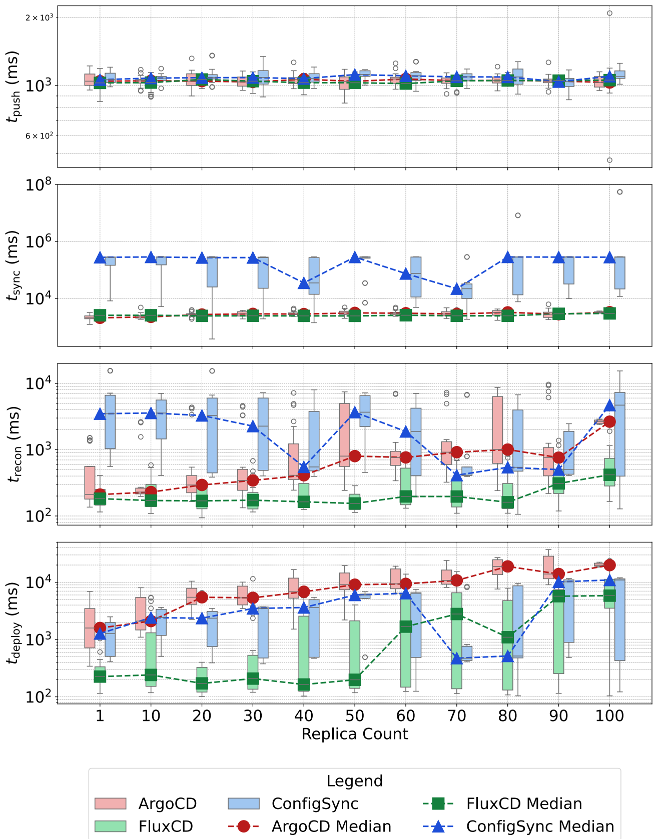


Fig. 3: Latency Comparison of GitOps Tools for Single App Deployment with Scaling Replica Count

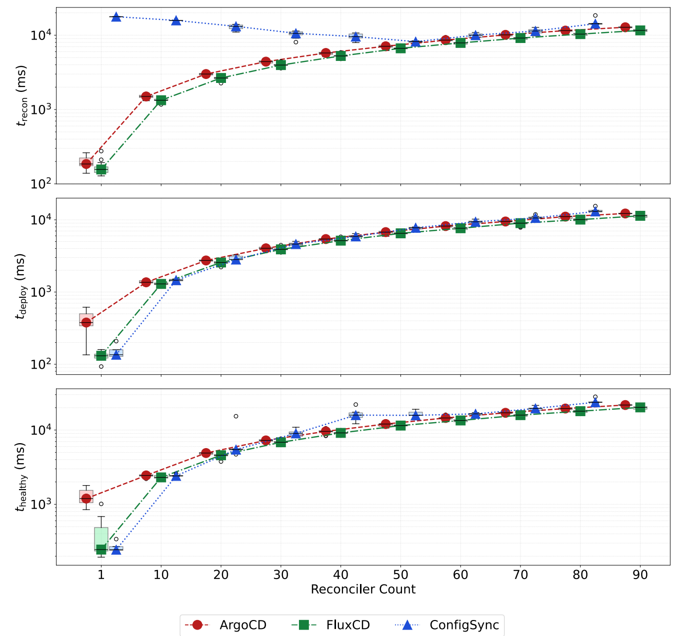


Fig. 4: Latency Comparison of GitOps Tools with Scaling Concurrent Multi-App Deployment

have kept $m = 90$ as the default maximum pod count of K3s clusters, which is 110 to accommodate additional reconcilers' control-plane pods.

### *A. Test-Case 1: Single-App Deployment Latency Comparison*

Fig. 3 depicts the combined measurements of the experiments. All three reconcilers show an identical $t_{\text{push}}$ of around $1\,\text{s}$, with outliers lying in a close neighbourhood, which is expected since $t_{\text{push}}$ is independent of the reconciliation process; however, this verifies the correctness of the tailing Push mechanism of the manifest generation. Argo CD and Flux CD outperform ConfigSync in $t_{\text{sync}}$ with identical latency bounded between 1–10 s, whereas ConfigSync took between 50–100 s. Examining the reason, we discovered that the *period* attribute of RootSync (i.e., ConfigSync's reconciler) is not persistent on the open-source version of ConfigSync if it runs outside of Google Cloud. Comparison of $t_{\text{recon}}$ shows Argo CD being the fastest (0.5–0.8 s), followed by Flux CD (0.5–5 s) both showing a predictable trajectory. However, ConfigSync shows significant fluctuation between 0.8–8 s. Finally, comparing $t_{\text{deploy}}$ shows ArgoCD performing better with median latency around $200\,\text{ms}$ up to $r = 50$ and it shoots up linearly to $10\,\text{s}$, converging with the latency of Flux CD and ConfigSync.

In summary, we observed that ArgoCD and FluxCD exhibit more consistent performance compared to ConfigSync. Notably, ArgoCD is slightly quicker in reconciliation and interfacing with the back-end Kubernetes cluster for deployment, compared to FluxCD.

### *B. Test-case 2A: Multi-App Latency Comparison*

Fig. 4 depicts the combined measurements of the experiments. All three reconcilers show an identical trend of $t_{\text{deploy}}$ and $t_{\text{healthy}}$. The $t_{\text{healthy}}$ result is as expected, as it is beyond the scope of the Reconcilers, similar to the case of $t_{\text{push}}$ in the testcase 1; therefore, we omit it in this testcase intentionally. We attribute the linear growth of $t_{\text{deploy}}$ to minor fluctuations resulting from parallelising the deployment of non-scaling applications, which Kubernetes schedulers take advantage of. However, there is a trade-off in resource utilisation, which is revealed in the next sub-section. After conducting the experiments several times, we observed a "V-Shaped" pattern emerging from $t_{\text{recon}}$ of ConfigSync. It starts with a $17.5\,\text{s}$ value, gradually converges with that of the other two at $r = 50$ to $7.5\,\text{s}$. This is roughly the same as its $t_{\text{recon}}$ for a single-app test case, and follows the same growth trajectory as Argo CD and Flux CD from there. At the time of writing this article, we don't have a clear explanation for this observation; however, we aim to investigate it further in our future work.

In summary, all reconcilers performed equally after concurrent deployment of the application with $r \geq 50$. Argo CD and Flux CD are the most consistent throughout, while ConfigSync shows convergence in cases of a large number of concurrent requests, with less fluctuation compared to Testcase 1.

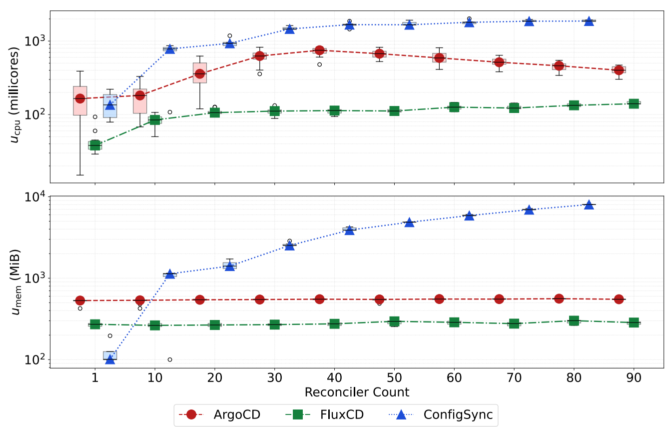


Fig. 5: Resource Utilisation Comparison of GitOps Tools with Scaling Concurrent Multi-App Deployment

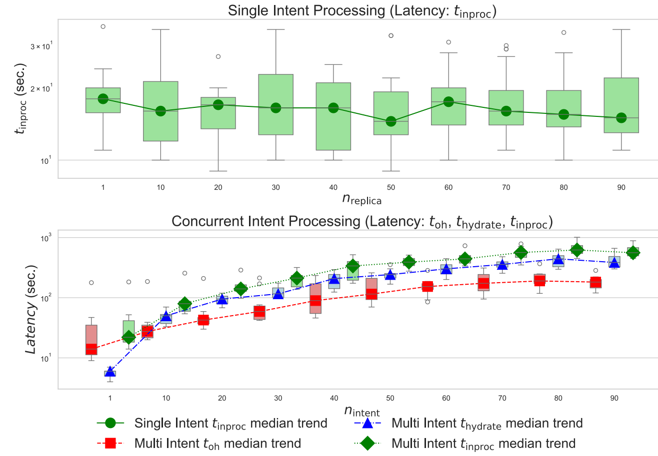


Fig. 6: Comparison of Nephio's Intent Processing Latency

### C. Test-Case 2B: Multi-App Resource Utilisation

For this investigation (Fig. 5), we measured the $u_{\text{cpu}}$ and $u_{\text{mem}}$ from K8s Container Runtime Interface. The median $u_{\text{cpu}}$ stays within 1–250 millicore for Flux CD, 150–750 millicore for Argo CD and 120–1900 millicore for ConfigSync. The median $u_{\text{mem}}$ remains consistent at 120 MB and 520 MB for Flux CD and Argo CD, respectively, with a significant difference from ConfigSync reaching up to 8 GB at $r = 90$. Our investigation suggests that the resource-intensive behaviour of ConfigSync is due to its concurrency handling mechanism. Unlike Flux CD and Argo CD, which run a single reconciler in their respective control plane namespaces (i.e., flux-system and argo, respectively), ConfigSync instantiates individual root-reconciler objects in the config-management-system namespace to bind each application, resulting in a significant overhead as the number of concurrent application requests scales.

In summary, Flux CD is the least resource-intensive, Argo CD has a similar memory footprint to Flux CD with higher CPU consumption, which settles down to a level comparable to Flux CD after $r = 40$, and ConfigSync is the most resource-intensive in both CPU and memory consumption.

### D. Test-Case 3: Nephio's Intent Processing Latency

Fig. 6 illustrates the latency profile of Nephio in processing the Intent ($t_{\text{intproc}}$) first, for a single deployment of Intent, scaling by replica; and second; for multiple deployment by scaling the number of Dry Packages and their corresponding PVs both within a range of $[1 : 90 : 10]$. The intent processing latency $t_{\text{intproc}} = t_{\text{hydration}} + t_{\text{oh}}$, where $t_{\text{oh}}$ is the latency introduced by the overhead processing including time to bring up the PVs, Webhook establishment between the PVs & BPR and discovery of Draft Packages in DPR. The $t_{\text{oh}}$ is proportional to the number of Intents. The experimental result shows that Nephio introduces a mean constant $t_{\text{intproc}}$ for Intent deployment, which is 17.62–23.85 s per Intent, including the default reconciliation period configured in Porch and the overhead of instantiating PV. In case of the multi-Intent experiments, we observed the mean $t_{\text{inproc}}$ =11.2 s with its components $t_{\text{oh}} = 6.16$ s and $t_{\text{hydrate}} = 4.95$ s. The mean $t_{\text{inproc}}$ for multi-Intent is slightly lower (by $\sim 6$ s) compared to that of the single-Intent because K8s instantiates the PVs simultaneously; therefore, on average, the latency due to overhead processing lowers as the number of Intents scales.

### E. Summary and Findings

Table III summarises the performance evaluation against all KPIs $K_{attr}$ from our experiments. Our findings show the following. First, the $t_{\text{sync}}$ & $t_{\text{recon}}$ of ConfigSync is higher and less deterministic compared to that of Argo CD and Flux CD. Second, Flux CD is more compute-intensive but faster than Argo CD. Third, Nephio's overall Intent processing latency with default Porch settings is almost constant per intent, and it scales linearly with the number of Intents.

To summarise each scenario, we processed the dataset as follows. First, we standardised the dataset concerning the scaling variable, i.e., the number of replicas for single intent and the number of apps for multi-intent use cases. Thereafter, we filter out the outliers by aggregating them based on their median value. For standard deviation $\sigma$, we computed the sample $\sigma$ after removing statistical outliers using the Interquartile Range (IQR). Specifically, any data point lying outside the interval $[Q_1 - 1.5 \cdot \text{IQR},\ Q_3 + 1.5 \cdot \text{IQR}]$ was excluded. The $\sigma$ was then calculated over the remaining values using the unbiased estimator.

## V. Conclusion & Future Scope

This work investigates the effect of integrating three GitOps operators (i.e., Argo CD, Flux CD, and ConfigSync) and an IBN orchestrator, Nephio, to enhance the E2E Intent Deployment performance of our bespoke orchestration platform, CAMINO. It describes a benchmarking pipeline with a methodology to collect defined KPIs that summarise the performance evaluation through latency and resource utilisation

TABLE III: Summary statistics of various latency & utilisation KPIs in single & multiple Intent deployment scenarios through GitOps tools (Argo CD, Flux CD & ConfigSync) and IBN Orchestrator (Nephio)

| **Tools** | **Single Intent Deployement** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $t_{\text{push}}$ (sec.) | | $t_{\text{sync}}$ (sec.) | | $t_{\text{recon}}$ (sec.) | | $t_{\text{deploy}}$ (sec.) | | | |
| | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | | |
| **Argo CD** | 1.05 | 0.01 | 2.83 | 0.37 | 0.01 | 0.01 | 9.07 | 0.04 | | |
| **Flux CD** | 1.04 | 0.01 | 2.58 | 0.09 | 0.0056 | 0.0034 | 0.02 | 0.02 | | |
| **Config Sync** | 1.01 | 0.02 | 217.53 | 112.15 | 0.03 | 0.05 | 0.11 | 0.01 | | |
| | **Multiple Intent Deployment** | | | | | | | | | |
| | $t_{\text{recon}}$ (sec.) | | $t_{\text{deploy}}$ (sec.) | | $t_{\text{healthy}}$ (sec.) | | $u_{\text{cpu}}$ (Milicore) | | $u_{\text{mem}}$ (MB) | |
| | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| **Argo CD** | 0.14 | 0.01 | 0.14 | 0.08 | 0.24 | 0.002 | 13.46 | 6.26 | 10.1 | 7.11 |
| **Flux CD** | 0.13 | 0.01 | 0.13 | 0.002 | 0.23 | 0.01 | 2.24 | 2.28 | 5.34 | 3.4 |
| **Config Sync** | 0.18 | 0.18 | 0.15 | 0.01 | 0.28 | 0.05 | 33.31 | 9.97 | 98.47 | 1.56 |
| **Nephio** | **Single Intent Deployment** | | | | **Multiple Intent Deployment** | | | | | |
| | $t_{\text{inproc}}$ (sec.) | | | | $t_{\text{inproc}}$ (sec.) | | $t_{\text{hydrate}}$ (sec.) | | $t_{\text{oh}}$ (sec.) | |
| | $\mu$ | | $\sigma$ | | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| | 17.9 | | 0.91 | | 7.79 | 0.34 | 4.97 | 0.42 | 2.73 | 0.13 |

in single and multi-intent deployment scenarios, supported by statistical analysis of experimental results.

In this benchmarking setup, we utilised a homogeneous intent deployment model by deploying a simple Nginx application to measure all KPIs; that said, we have established a configurable testbed infrastructure. Hence, as an extension, we shall leverage it to conduct benchmarking in a heterogeneous setup with a variety of standard Network Functions (e.g., 5G Core). In addition, we aim to advance this framework as a tool to generate large-scale datasets to train machine-learning models, enabling the prediction of anticipatory latency and resource consumption in an intent-deployment scenario. AI-Native IBN for a 6G communication system would leverage such predictions for optimising proactive resource allocation.

## VI. Acknowledgement

This work is funded in part by REASON+, a UK Government-funded project under the Future Open Networks Research Challenge (FONRC), sponsored by the Department of Science, Innovation, and Technology (DSIT), and in part by Innovate UK funding under the CelticNext SUSTAINET-guarDian project.

## References

[1] A. Leivadeas and M. Falkner, "A Survey on Intent-Based Networking," *IEEE Commun. Surv. Tutor*, vol. 25, no. 1, pp. 625–655, 2023.

[2] F. Beetz and S. Harrer, "GitOps: The Evolution of DevOps?" *IEEE Soft.*, vol. 39, no. 4, pp. 70–75, 2022.

[3] L. Foundation, "Nephio," https://nephio.org, Nephio Project, Tech. Rep., 2025, accessed: 2025-07-19.

[4] L. Bonati, M. Polese, S. D'Oro, P. B. del Prever, and T. Melodia, "5G-CT: Automated Deployment and Over-the-Air Testing of End-to-End Open Radio Access Networks," *IEEE Comms. Mag.*, vol. 63, no. 1, pp. 155–160, 2025.

[5] P. Wörndle, S. Terrill, and T. Dinsing, "Automating telecom software deployment with GitOps," *Ericsson Technology Review*, vol. 2023, no. 2, pp. 2–10, 2023.

[6] GitLab Inc., "GitLab," https://about.gitlab.com, 2025, accessed: 2025-07-19.

[7] Argo CD, "Argo CD - Declarative GitOps CD for Kubernetes," https://argoproj.github.io, 2025, accessed: 2025-07-19.

[8] Flux CD, "Flux CD," https://fluxcd.io, 2025, accessed: 2025-07-19.

[9] Google, "Config Sync," https://github.com/GoogleContainerTools/kpt-config-sync, 2025, accessed: 2025-07-19.

[10] K. Antonakoglou, I. Mavromatis, S. Ghosh, M. Rouse, and K. Katsaros, "CAMINO: Cloud-Native Autonomous Management and Intent-Based Orchestrator," in *Proc. of EuCNC/6G Summit*, 2025, pp. 357–362.

[11] Ramadoni, E. Utami, and H. A. Fatta, "Analysis on the Use of Declarative and Pull-based Deployment Models on GitOps Using Argo CD," in *Proc. of Int. Conf. on ICOIACT*, 2021, pp. 186–191.

[12] S. Gupta, M. Bhatia, M. Memoria, and P. Manani, "Prevalence of GitOps, DevOps in Fast CI/CD Cycles," in *Proc. of Int. Conf. COM-IT-CON*, vol. 1, 2022, pp. 589–596.

[13] R. López-Viana, J. Díaz, and J. E. Pérez, "Continuous Deployment in IoT Edge Computing : A GitOps implementation," in *Proc. of Int. Conf CISTI*, 2022, pp. 1–6.

[14] T. Kormaník and J. Porubän, "Exploring GitOps: An Approach to Cloud Cluster System Deployment," in *Proc. of Int. Conf. ICETA*, 2023, pp. 318–323.

[15] A. Leiter, A. Hegyi, I. Kispal, P. Boosy, N. Galambosi, and G. Z. Tar, "GitOps and Kubernetes Operator-based Network Function Configuration," in *Proc. of IEEE/IFIP*, 2023, pp. 1–5.

[16] M. Vitumbiko and Y. Kim, "Design of network setup automation using gitops operation," in *Proc. of Int. Conf. ICOIN*, 2025, pp. 402–405.

[17] R. Shrestha and A. A. Nur Ali, "Configuration Management in Kubernetes Environments: A GitOps Approach," in *Proc. of IEEE/ACM UCC*, 2024, pp. 497–502.

[18] Red Hat, Inc., "Ansible documentation," https://docs.ansible.com/ansible/latest/index.html, 2025, accessed: 2025-08-11.