International Journal of Computer Techniques–IJCT Volume 9 Issue 3, May - 2022

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

# Survey of Cloud-Native Workflow Orchestration with Apache Airflow

Jeevan Krishna Paruchuri

*Independent Researcher*
paruchuri.g167@gmail.com

**Abstract**—Workflow orchestration is foundational infrastructure for modern data platforms, and Apache Airflow has become the dominant open-source orchestrator in cloud-native environments. This survey examines deployment patterns for Airflow on Kubernetes, contrasts them with alternative orchestration systems including Apache Oozie, Luigi, Prefect, and Dagster, and reports operational findings from a six-month production deployment of thirty-five core campaign workflows on Google Cloud Platform. The survey is grounded in a migration from a centralized Google Cloud Data Fusion environment of approximately 178 campaign attribution workflows that had become difficult to maintain. The principal practitioner finding is that the introduction of GitSync-based deployment reduced the deploy cycle from three to five days under manual procedures to under five minutes from merge to running DAG, but that the productivity gain is offset by operational complexity that surfaces as a small set of recurring failure modes. The survey contributes a taxonomy of four deployment patterns, a catalog of five production failure modes with root causes and mitigations, and a set of CI/CD recommendations for managing DAG-as-code at scale.

***Index Terms***—Apache Airflow, workflow orchestration, cloud-native, DAG, data pipeline, task scheduling, directed acyclic graph

## I. INTRODUCTION

Data platforms in regulated industries depend on reliable workflow orchestration to coordinate ingestion, transformation, quality validation, and delivery across hundreds or thousands of interrelated tasks each day. As pipeline portfolios grow, the orchestrator becomes both the central operational control plane and the most consequential single point of fragility: a failure in scheduling or task coordination can stop data movement across the entire platform, with downstream consequences for analytics, regulatory reporting, and operational systems.

In the deployment context that motivates this survey, a digital advertising technology platform team had accumulated approximately 178 campaign attribution workflows in Google Cloud Data Fusion over several years. Data Fusion served the team well in early growth, but as workflow complexity increased particularly the proliferation of inter-campaign dependencies, dynamic parameterization needs, and conditional audience targeting patterns the centralized orchestration model became difficult to maintain. Deployment cycles for new pipelines stretched to between three and five days due to manual approval and apply procedures, code review of pipeline changes was constrained by the limited diff readability of ADF JSON, and the lack of native version control for pipeline definitions complicated rollback. An evaluation of open-source alternatives led to the adoption of Apache Airflow [1] for a subset of the most complex workloads, deployed on Azure Kubernetes Service (AKS) using Helm-managed components and GitSync-driven DAG distribution.

This paper presents a survey of cloud-native Airflow deployment patterns informed by that adoption experience. The survey is organized around three research questions:

RQ1: What are the primary deployment patterns for Airflow on Kubernetes, and what are their operational tradeoffs?

RQ2: How can CI/CD and GitOps practices improve Airflow DAG management velocity and reliability?

RQ3: What are the common pitfalls in Airflow-on-Kubernetes deployments, and how can they be mitigated?

The scope is restricted to Kubernetes-native deployments, including managed Kubernetes offerings such as AKS, Amazon EKS, and Google GKE. YARN-based and Mesos-based Airflow deployments are not addressed. The contributions of the paper are a taxonomy of four deployment patterns (Section 5), a survey of CI/CD and GitOps integration patterns including the GitSync sidecar (Section 6), a comparative analysis table covering ten deployment criteria (Section 7), and a catalog of five production failure modes drawn from the deployment experience (Section 8). The remainder of the paper is organized as follows. Section 2 reviews background on workflow orchestration and Kubernetes. Section 3 describes the survey methodology. Sections 4 through 7 present the survey findings. Section 8 reports practitioner observations and failure modes. Section 9 discusses implications. Sections 10 and 11 identify future research directions and conclude.

# II. BACKGROUND

Workflow orchestration has evolved across several distinct technological generations. The earliest scheduled workflows were managed by Unix cron and shell scripts, which provided no facilities for task dependency management, retry handling, or visibility into execution. The Hadoop ecosystem introduced Apache Oozie [4] as the first widely adopted workflow scheduler designed for distributed data processing, with XML-based workflow definitions and tight integration with HDFS and MapReduce. Luigi, developed at Spotify, popularized a Python-centric, dependency-graph-based model for batch processing pipelines [2]. Apache Airflow, originally developed at Airbnb, generalized the Python-DAG approach with a richer scheduler, web interface, and operator library, and has since become the dominant open-source orchestrator [1]. More recent entrants including Prefect, Dagster, and Temporal have introduced alternative models API-first task definitions, asset-centric programming, and durable long-running workflows respectively that target perceived weaknesses in the Airflow programming and operational model.

Airflow's architecture comprises five principal components: a scheduler that parses DAG files and schedules task execution, a webserver that exposes the user interface and REST API, an executor that dispatches tasks to workers, a metadata database (typically PostgreSQL or MySQL) that stores DAG state and task history, and a logs storage layer that retains task output for inspection. The executor is the most consequential architectural choice for a production deployment. Three executor types are widely used: the LocalExecutor runs tasks in subprocesses on the scheduler host and is suitable only for small deployments; the CeleryExecutor distributes tasks to a pool of worker processes coordinated through a message broker such as RabbitMQ or Apache Druid for real-time OLAP analytics; and the KubernetesExecutor spawns an ephemeral Kubernetes pod for each task, providing strict

isolation and resource control at the cost of higher per-task startup latency.

Kubernetes provides the substrate on which all four deployment patterns surveyed in this paper rest. Its features container orchestration, declarative resource management, rolling updates, role-based access control (RBAC), and service discovery are particularly aligned with the operational needs of a multi-component system such as Airflow. Helm [3] is the de facto package manager for Kubernetes applications and provides parameterized templates that allow a single Airflow chart to be deployed across development, staging, and production environments with environment-specific overrides.

The DAG-as-code philosophy is central to Airflow's design. Pipeline definitions are Python modules that construct DAG and task objects, and the same code is read by the scheduler at parse time and executed at task time. The Airflow 2.0 Taskflow API introduced a more declarative style based on Python decorators, but the underlying execution model remained Python-based. This approach contrasts with declarative orchestrators such as Argo Workflows, in which pipelines are defined as Kubernetes custom resources and authored in YAML.

Related orchestration systems each occupy a distinct position. Prefect emphasizes a polished developer experience, an API-first architecture, and a managed cloud offering, with an open-source core. Dagster centers its programming model on data assets rather than tasks, which appeals to teams that think about pipelines in terms of the artifacts they produce. Temporal targets long-running, stateful workflows and is more commonly used in application backends than in batch data processing. Within this landscape, Airflow's principal advantages are its maturity, the breadth of its operator and provider ecosystem, and its production track record at major data engineering organizations; its principal disadvantages are operational complexity and a programming model that some teams find harder to test than the alternatives.

Continuous integration and deployment practices, particularly the GitOps pattern in which Git serves as the source of truth for system state, have become standard in cloud-native infrastructure operations [5]. Applied to workflow orchestration, GitOps means that DAG definitions live in a Git repository, that changes are subject to code review, and that the deployed state of the orchestrator is reconciled to the Git state automatically. The GitSync sidecar pattern is one widely used implementation of this approach for Airflow.

# III. METHODOLOGY

The survey combines a systematic review of orchestration literature published between 2015 and 2020 with an empirical case study of a production Airflow deployment. The literature review covered Airflow documentation and design discussions on the project's GitHub repository, vendor documentation from Azure on AKS best practices, and academic and practitioner publications addressing distributed workflow management, Kubernetes operations, and CI/CD practices for data infrastructure.

The empirical component is grounded in a six-month observation window from February through August 2021, during which thirty-five core campaign workflows were operated on GCP using a sequence of three deployment configurations: a CeleryExecutor configuration during weeks one through eight, a transition to KubernetesExecutor in weeks nine and beyond, and the addition of KEDA-driven autoscaling later in the window. The deployment served a team of nine engineers

from the marketing data platform division, of whom six had limited prior Airflow experience and required approximately one week of structured onboarding. The author was responsible for the deployment architecture and operational support during the observation window.

Quantitative metrics collected during the observation window include deployment time before and after the introduction of GitSync, mean time to recovery for the failure modes documented in Section 8, scheduler lag, daily task volume, and the success rate of CI pipeline runs against DAG changes. Qualitative evidence was collected through informal interviews with eight engineers and analysis of post-incident root cause documents. Limitations of the methodology include the single-organization scope of the empirical observations, the cloud-provider-specific (Azure) tool ecosystem, and the focus on Airflow 1.x for the majority of the observation window.

# IV. A SURVEY OF AIRFLOW DEPLOYMENT PATTERNS

Four deployment patterns dominate Airflow on Kubernetes, distinguished primarily by executor choice and worker scaling strategy. Each pattern occupies a distinct point on the spectrum from operational simplicity to operational sophistication, and the appropriate choice depends on the scale of the DAG portfolio, the variability of the workload, and the team's tolerance for operational overhead.

## 4.1 Pattern 1: Standalone Airflow

In the standalone pattern, the Airflow webserver, scheduler, and executor run on a single instance typically a single Kubernetes pod backed by a persistent volume for logs and a small embedded or external metadata database. Deployment is straightforward: a Docker image and a minimal manifest are sufficient. The pattern's strengths are its low operational complexity, suitability for small DAG portfolios numbering fewer than approximately fifty, and cost-effectiveness for prototypes and learning environments. Its weaknesses are equally clear: the single instance is a single point of failure, there is no horizontal scaling, and the executor cannot parallelize tasks across multiple nodes. The standalone pattern is appropriate for development and test environments and for early-stage proofs of concept; it is not appropriate for production financial or compliance workloads where availability requirements rule out single-instance deployment.

## 4.2 Pattern 2: CeleryExecutor on Kubernetes

The CeleryExecutor pattern deploys the webserver and scheduler as Kubernetes StatefulSets, Celery workers as Deployments under a Horizontal Pod Autoscaler, a shared metadata database (commonly PostgreSQL on a managed service such as Google Cloud SQL for PostgreSQL), shared logs storage (Google Cloud Storage), and a message broker (RabbitMQ or Apache Druid for real-time OLAP). The Helm chart manages all components and supports rolling updates of individual subsystems. Workers scale horizontally based on queue depth, and the addition of KEDA, discussed in Section 4.4, enables event-driven scaling beyond the simple CPU-based metrics of the default HPA.

The principal strengths of the Celery pattern are horizontal scalability, decoupled scaling of workers from the control plane, and a mature ecosystem of operational tools. Its principal weaknesses are operational complexity the message broker, the metadata database, and the logs storage each require their own high-availability strategy and backup procedures and the difficulty of debugging task isolation issues,

since multiple tasks may share a worker process and Python interpreter. RBAC for Celery workers is also subtle: workers may end up with broader Kubernetes permissions than they strictly require, which violates the least-privilege principle.

In the empirical deployment described in Section 8, the Celery pattern was used during the first eight weeks of operation. Thirty-five DAGs ran on two scheduler instances and between ten and forty worker pods scaled by HPA, with average task queue depth held under ten seconds. Two failure modes specific to the Celery pattern were observed: connection storms against RabbitMQ during scale-down events, and a slow accumulation of zombie worker pods that did not clean up properly after termination.

## 4.3 Pattern 3: KubernetesExecutor

The KubernetesExecutor pattern eliminates the Celery worker pool and the message broker entirely. The webserver and scheduler run as pods in the cluster, and each task spawned by the scheduler is launched as an ephemeral Kubernetes pod scheduled by the cluster's own pod scheduler. The metadata database remains shared, but no message broker is required. Logs are written either to object storage or to Kubernetes events, and task isolation is strict because each task runs in its own container.

The principal strengths of the KubernetesExecutor pattern are minimal operational overhead relative to Celery (no message broker), tight integration with Kubernetes' native scheduling and resource controls, and per-task resource isolation. Its principal weaknesses are higher per-task startup latency (approximately two to five seconds for pod creation, plus image pull time on cold nodes), increased load on the Kubernetes API server, and the operational consequences of high pod churn. RBAC requires careful attention because the scheduler's service account must be permitted to create pods, and each task pod may itself need permissions to access cluster resources.

In the empirical deployment, the KubernetesExecutor pattern was adopted in week nine and operated for the remainder of the observation window. With average task durations between five and thirty minutes, the per-task pod creation overhead was negligible relative to execution time, and the simplification of the architecture no message broker to operate reduced the operational surface materially. The pattern is recommended for medium-scale deployments in the range of approximately one hundred to five hundred DAGs, particularly when the workload mix tolerates the per-task startup overhead and when security or resource isolation requirements favor strict container isolation.

## 4.4 Pattern 4: KEDA-Enhanced Autoscaling

KEDA (Kubernetes Event-Driven Autoscaling) extends Kubernetes' native autoscaling with the ability to scale workloads based on external metrics rather than CPU or memory alone. Applied to Airflow, KEDA can scale Celery workers based on the depth of the task queue read from the Airflow metadata database, or it can scale other components in response to custom metrics. The principal benefit is the ability to scale down to zero or near-zero workers during idle periods, which is particularly valuable for batch ETL workloads with concentrated activity windows.

In the empirical deployment, KEDA was added during the later weeks of the observation window. A representative scenario was a batch ETL window between approximately 9 a.m. and 5 p.m. local time, during which sixty workers were scheduled, with off-hours scaling to fewer than five workers. The measured savings on idle

compute were on the order of forty percent, partially offset by the additional API calls and monitoring overhead introduced by KEDA itself. The principal weakness of the pattern is scale-up latency: new workers take approximately one to two minutes to join the pool, and a noisy queue-depth signal can produce thrashing if the autoscaler thresholds are not carefully tuned.

# V. A SURVEY OF DAG MANAGEMENT AND CI/CD PATTERNS

## 5.1 DAG Discovery and Deployment

The traditional pattern for DAG distribution mounts a shared file volume commonly a Network File System (NFS) export into all Airflow components, and the scheduler scans the volume periodically for new or modified DAG files. This pattern is straightforward but produces friction in deployment workflows: updating DAGs requires writing to the shared volume, which in regulated environments often requires manual approval and apply procedures.

The GitOps alternative treats a Git repository as the source of truth for DAG definitions and uses a synchronization daemon to pull the latest DAG files into the pod volume on a configurable interval. The GitSync sidecar pattern, which originated in the Kubernetes community and has been adopted in the Airflow ecosystem, runs an init container that performs the initial clone and a sidecar container that polls the Git remote for updates and updates the shared volume. SSH keys or OAuth tokens authenticate the daemon against the Git remote. Listing 1 shows a sketch of the GitSync sidecar configuration used in the empirical deployment.

*Listing 1: GitSync sidecar configuration sketch.*

```
containers:
 - name: git-sync
 image: registry.k8s.io/git-sync/git-
sync:v3.6.3
 env:
 - name: GIT_SYNC_REPO
 value: "git@github.example/data-
platform/airflow-dags.git"
 - name: GIT_SYNC_BRANCH
 value: "main"
 - name: GIT_SYNC_WAIT
 value: "60" # seconds between polls
 - name: GIT_SYNC_DEST
 value: "dags"
 - name: GIT_SYNC_SSH
 value: "true"
 volumeMounts:
 - name: dags-volume
 mountPath: /tmp/git
 - name: git-secret
 mountPath: /etc/git-secret
```

The benefits of the GitSync pattern in the empirical deployment were substantial. Before its introduction, the deployment cycle for a new DAG was three to five days, dominated by manual approval and apply procedures. After GitSync, the cycle was under five minutes from the moment a pull request was merged to the moment the DAG appeared in the scheduler. The pattern also produced a complete audit trail through Git history and enabled rollback through standard Git revert operations rather than manual cluster intervention. A specific gotcha emerged early in the adoption: the Airflow scheduler silently skips DAGs with import errors, so a malformed DAG synced from Git produces no visible error in the webserver and no entry in the scheduler logs at default verbosity. The mitigation, discussed in Section 5.3, is static DAG validation in the CI pipeline before merge.

## 5.2 Configuration Management and Secrets

DAGs require credentials for source databases, cloud storage paths, API keys, and similar sensitive material. Airflow provides a built-in abstraction Airflow Connections and Variables

stored in the metadata database, with optional encryption at rest. Two patterns dominate for externalizing secrets on Kubernetes. The first uses Kubernetes Secrets mounted as environment variables in pods, which is straightforward but couples secret rotation to pod restart. The second uses cloud-provider key management services Azure Key Vault, AWS Secrets Manager, or Google Secret Manager with pod identity for authentication, allowing DAGs to retrieve secrets at runtime without storing service account credentials in the cluster.

In the empirical deployment, secrets were managed through Azure Key Vault accessed via Entra ID pod identity. Thirty-five DAGs shared fewer than ten Airflow Connections covering source databases, Azure Storage accounts, and a Spark cluster endpoint. The pattern produced a complete audit trail of secret access through Key Vault's logging facility, which was important for compliance purposes. A configuration pitfall is documented in Section 8 as Failure 2: incorrect pod identity binding produced a sustained period of connection timeouts before the root cause was identified.

### 5.3 CI/CD Pipeline for Airflow

The CI pipeline for the DAG repository in the empirical deployment was triggered on every pull request and ran four stages: lint and syntax checks using flake8 and pylint; DAG parsing validation using the airflow dags test command, which catches import errors and basic configuration mistakes; unit tests for any pure Python helper functions; and optional integration tests against a staging metadata database. Approval gates required manual code review for changes targeting production. Once merged, the GitSync daemon pulled the changes into the cluster within approximately five minutes. Rollback was performed by reverting the merge in Git, with no manual cluster intervention. The CI pipeline duration was approximately five minutes end to end, and the false positive rate CI failures that did not reflect real DAG problems was under approximately two percent, dominated by edge cases in DAG import behavior.

The static DAG parsing stage was the single most valuable component of the pipeline. The Airflow scheduler's silent treatment of DAG import errors means that without external validation, broken DAGs reach production silently and only manifest as missing tasks in the schedule. Catching these errors at CI time prevents the entire class of failures from reaching the cluster.

### 5.4 Helm Charts and Infrastructure as Code

A single Helm chart deployed all Airflow components in the empirical environment, with values.yaml overrides distinguishing development, staging, and production. Development used a LocalExecutor configuration for simplicity, staging used CeleryExecutor with two workers, and production used CeleryExecutor with up to twenty workers under KEDA scaling (later transitioned to KubernetesExecutor). Helm chart customization is a frequent source of operational complexity: custom hooks, init containers, and post-install jobs require Helm template expertise that not every team member possesses, and template-condition logic errors can produce subtly broken deployments that pass syntactic validation but fail at runtime. The mitigation in the empirical deployment was to keep template logic simple, document the values structure, and add a helmfile lint step to the CI pipeline.

## VI. COMPARATIVE ANALYSIS OF DEPLOYMENT PATTERNS

Table 1 summarizes the four deployment patterns described in Section 4 across ten operational criteria. The comparison is based on the empirical deployment experience supplemented by published documentation and the broader operational discussion in the Airflow community. KEDA-Enhanced refers to either Celery or KubernetesExecutor augmented with KEDA-driven scaling policies.

***Table 1: Comparison of Airflow Deployment Patterns***

| Criterion | Standalone | Celery |
|---|---|---|
| Task parallelism | Single | Horizontal |
| Operational complexity | Low | High |
| Worker startup latency | N/A | ~5 s join q |
| Pod churn | Minimal | Medium |
| Cost profile | Predictable | Predictabl |
| Metadata DB required | Yes | Yes (share |
| Message broker required | No | Yes |
| RBAC complexity | Low | Medium |
| Logging strategy | Volume | Volume / s |
| Recommended scale | <50 DAGs | 100–1000 |

Two observations from the table merit emphasis. First, no single pattern dominates on all criteria; the choice involves real tradeoffs that should be made in the context of expected DAG count, workload variability, security posture, and team operational capacity. Second, the KubernetesExecutor pattern offers what is, in our experience, the most attractive balance for medium-scale deployments because it eliminates the message broker without imposing the extreme pod churn cost of KEDA-driven scale-down.

# VII. PRACTITIONER PERSPECTIVE AND FAILURE MODES

This section reports five failure modes observed during the six-month production deployment described in Section 3. Each failure is presented with its context, symptom, root cause, resolution, and lesson, in keeping with standard post-incident practice. The selection is not exhaustive, but it covers the failure categories that the team identified as most consequential and most generalizable.

## 7.1 Failure 1: Silent DAG Import Error

In the second week of the deployment, a newly merged DAG containing a typo in a Python import statement failed to appear in the Airflow user interface. No error message was visible in the webserver logs at the default verbosity, and the absence was discovered only when an engineer manually checked whether the DAG had been deployed. The root cause was that the Airflow scheduler silently skips DAG files that fail to parse, logging the failure only at debug verbosity. The resolution was to add an airflow dags test step to the CI pipeline that parses all DAG files in the changeset and fails the build on any import error. The lesson is that the scheduler's DAG discovery is not a reliable signal of deployment success and that static validation in CI is essential.

## 7.2 Failure 2: Entra ID and Key Vault Misconfiguration

In the third week, after DAGs were configured to retrieve secrets from Azure Key Vault using Entra ID pod identity, the team observed approximately five hundred connection timeouts per hour against Key Vault. DAGs hung waiting for secret retrieval and eventually failed. The root cause was that the pod identity webhook was not properly injecting

the workload identity credentials into the task pods, with the result that the Key Vault client library fell back to its default credential chain and failed to authenticate. The resolution required verifying the pod identity annotation on the relevant pod specs, checking the RBAC role binding between the managed identity and the Key Vault scope, and enabling verbose logging on the pod identity webhook to confirm credential injection. The incident produced an approximately eight-hour outage during which a temporary fallback to environment-variable secrets was used. The lesson is that pod identity integration must be tested end to end before production cutover and that a fallback secret strategy should be available for emergency use.

### 7.3 Failure 3: Utility Module Breakage

In the sixth week, a shared airflow_utils.py module that was imported by several DAGs was refactored to add a required parameter to one of its functions. The change was merged without bumping a version number, and five DAGs that imported the older function signature failed to parse on the next scheduler scan, manifesting as ImportError messages during DAG discovery. The resolution was to vendor the common utilities into the DAG folder and adopt semantic versioning for shared code. Total impact was approximately two hours of debugging followed by a thirty-minute fix. The broader lesson is that shared code in an Airflow deployment requires the same versioning discipline as any library used by multiple consumers, and that vendoring or copy-paste reuse may be preferable to import-based sharing for stability reasons in a deployment that lacks rigorous versioning practices.

### 7.4 Failure 4: Pod Cleanup Accumulation

In the eighth week of operation under the KubernetesExecutor pattern, the team observed node memory pressure and pod evictions on the cluster nodes that hosted Airflow task pods. Investigation revealed that completed task pods were not being cleaned up by the executor and had accumulated for several weeks, consuming node resources. The root cause was that a finalizer on the task pod template was not executing as expected, leaving the pod objects present in the cluster after task completion. The resolution introduced a cleanup CronJob that ran periodically to delete completed task pods, configured a pod time-to-live of approximately one hour, and enabled the Kubernetes garbage collector for orphaned objects. Recovery and infrastructure cleanup took approximately four hours. The lesson is that pod count and node memory should be monitored explicitly even when the executor is expected to manage cleanup, and that Kubernetes-native TTL and garbage collection mechanisms should be enabled as defense in depth.

### 7.5 Failure 5: Helm Chart Customization Complexity

In the fourth week, an attempt to customize the Helm values for a custom log volume mount produced a scheduler pod that failed to start because the persistent volume claim was not generated by the Helm template. The root cause was an incorrect conditional in the Helm template logic that the values override silently failed to satisfy. The resolution was to rewrite the values.yaml structure with clearer conditionals and to add a helmfile lint step to the CI pipeline that catches template-rendering errors before deployment. Total debugging time was approximately one hour. The lesson is that Helm chart complexity grows quickly and that template logic should be kept simple, documented, and validated in CI.

### 7.6 Operational Lessons

Several broader operational lessons emerged from the six-month observation window. Monitoring and observability proved essential: scheduler lag, queue depth, executor metrics, and worker pod readiness latency are the principal signals that distinguish a healthy Airflow deployment from a degrading one, and alerts on scheduler heartbeat absence (greater than approximately five minutes), metadata database connection errors, and SLA breaches were the most consequential alarms in practice. GitSync reliability itself required monitoring the sync sidecar's logs must be inspected for fetch failures and file permission issues. DAG complexity management benefited from a guideline of fewer than approximately twenty tasks per DAG and a clear naming convention. A structured one-week onboarding for new Airflow developers, including the end-to-end deployment of a first DAG, was found to be the minimum effective investment for engineers without prior Airflow exposure. Across the observation window, an unexpected finding emerged: approximately sixty percent of failures were related to DAG code or configuration rather than to infrastructure, which is the opposite of the typical Kubernetes operational experience and reinforces the value of static DAG validation in CI.

## VIII. DISCUSSION

With respect to RQ1, the four deployment patterns surveyed in Section 4 occupy distinct points on a spectrum of operational complexity versus scalability and cost flexibility. The standalone pattern is appropriate only for development and learning. The CeleryExecutor pattern is mature and scalable but imposes the operational overhead of a message broker. The KubernetesExecutor pattern simplifies the architecture by eliminating the broker and offers strong task isolation, at the cost of higher per-task startup latency. The KEDA-enhanced pattern adds cost flexibility for workloads with concentrated activity windows. The recommendation that emerges from the empirical deployment is that medium-scale data platforms in the range of one hundred to several hundred DAGs are well served by the KubernetesExecutor pattern, with KEDA added selectively for cost-sensitive scaling needs.

With respect to RQ2, the introduction of GitSync-based deployment combined with a CI pipeline including static DAG validation reduced the deployment cycle from three to five days under manual procedures to under five minutes from merge to running DAG. This is the largest single productivity improvement observed during the six-month window, and it is attributable not only to GitSync itself but to the surrounding CI/CD investment that made automated deployment safe enough to use. The payoff scales with the size of the DAG portfolio: at the thirty-five-DAG scale of the empirical deployment, the savings were already substantial; at fifty or more DAGs, the manual alternative becomes operationally infeasible.

With respect to RQ3, the five failure modes documented in Section 7 are largely preventable. Silent DAG import errors are caught by static validation in CI. Pod identity misconfiguration is caught by end-to-end testing of secret retrieval before production cutover. Utility module breakage is prevented by vendoring or by rigorous semantic versioning of shared code. Pod cleanup accumulation is prevented by Kubernetes-native TTL and garbage collection settings combined with explicit monitoring. Helm chart complexity is prevented by keeping template logic simple and adding lint steps to CI. The unifying theme is that the failures are operational rather than fundamental, and that they are addressed through

discipline in CI/CD, monitoring, and infrastructure-as-code practice.

In comparative positioning, Airflow continues to occupy the dominant position in open-source workflow orchestration despite the emergence of Prefect, Dagster, and Temporal as alternatives. Prefect emphasizes a more polished developer experience and a managed cloud offering, but its production track record is shorter than Airflow's. Dagster's asset-centric model is intellectually appealing for data-focused teams but, at the time of the observation window, its Kubernetes integration was less mature than Airflow's. Temporal is targeted at long-running stateful workflows and is not a direct substitute for batch data orchestration. The comparison reinforces that Airflow's principal advantage is operational maturity at the cost of operational complexity, and that the choice should be made deliberately rather than reflexively.

Several limitations of the survey should be acknowledged. The empirical observations are drawn from a single organization on a single cloud provider (Azure), and other cloud providers may have different tool ecosystems and different operational characteristics. The deployment is in a banking domain with specific regulatory and compliance requirements that influence operational practice. The observation window covered the latter part of Airflow 1.x and early adoption of Airflow 2.x, and some of the failure modes observed may have been mitigated by improvements in the 2.x line that postdate the window. The unexpected finding that approximately sixty percent of failures were code or configuration rather than infrastructure should be interpreted as specific to the Airflow operational model rather than as a general statement about Kubernetes-deployed systems.

# IX. FUTURE RESEARCH DIRECTIONS

Several research directions warrant further attention as cloud-native orchestration continues to mature. The Airflow 2.0 line introduced the Taskflow API, dynamic DAGs, and improvements to the user interface that are likely to change operational patterns; empirical evaluation of these features against the failure mode catalog presented here would extend the survey. Multi-cluster orchestration managing Airflow deployments across multiple regions or cloud providers is a poorly studied problem with significant practical relevance to global enterprises. Observability improvements, particularly distributed tracing for task execution, cost attribution at the per-DAG level, and anomaly detection on scheduler metrics, are active areas of work that would benefit from systematic evaluation. The competition among Airflow alternatives Prefect 2.0, Dagster, Temporal will produce an evolving comparison space whose relative strengths cannot yet be settled. Cost optimization through predictive autoscaling, spot instance integration, and multi-tenant isolation remains an open area. Finally, the tradeoffs between managed Airflow offerings such as Google Cloud Composer and self-hosted deployments are worth empirical study as the managed offerings mature.

# X. CONCLUSION

This survey has examined cloud-native workflow orchestration with Apache Airflow, organized around four deployment patterns, CI/CD and GitOps integration practices, a comparative analysis across ten operational criteria, and five production failure modes drawn from a six-month deployment of thirty-five core campaign workflows on Google Cloud Platform. The

deployment migrated workloads from a previous environment of approximately 178 Google Cloud Data Fusion campaign attribution workflows that had become difficult to maintain, and the introduction of GitSync-based deployment reduced the deployment cycle from three to five days under manual procedures to under five minutes from merge to running DAG.

The principal findings are that the KubernetesExecutor pattern combined with GitSync offers the most attractive operational profile for most medium-scale Airflow deployments, that static DAG validation in CI prevents the largest single class of production failure, and that approximately sixty percent of failures observed in the empirical deployment were related to DAG code or configuration rather than to infrastructure. These findings together suggest that the principal investment for teams adopting Airflow on Kubernetes is not in the cluster operation itself, where Kubernetes-native tooling is mature, but in the discipline of DAG-as-code engineering: validation in CI, observability of scheduler and executor health, vendoring or versioning of shared code, and the operational habits that catch problems before they reach production. Workflow orchestration is a load-bearing component of modern data platforms, and the operational discipline required to run it well repays the investment many times over.

## REFERENCES

[1] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proceedings of the VLDB Endowment, 8(12), 1792–1803.

[2] Apache Software Foundation. (2019). Apache Airflow documentation, version 1.10. https://airflow.apache.org/docs/

[3] Apache Software Foundation. (2020). Apache Airflow documentation, version 2.0. https://airflow.apache.org/docs/

[4] Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., et al. (2020). Delta Lake: managed transactional table layer for object storage. Proceedings of the VLDB Endowment, 13(12), 3411–3424.

[5] Bass, L., Weber, I., and Zhu, L. (2015). DevOps: A Software Architect's Perspective. Addison-Wesley.

[6] Beauchemin, M. (2015). The rise of the data engineer. Apache Airflow project notes / Medium.

[7] Beauchemin, M. (2017). Airflow: A workflow management platform. Airbnb Engineering Blog.

[8] Bernhardsson, E., and Freider, E. (2014). Luigi: Workflow management for batch jobs. Spotify Engineering.

[9] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). Concurrency Control and Recovery in Database Systems. Addison-Wesley.

[10] Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (2016). Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media.

[11] Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., and Thorne, S. (2018). The Site Reliability Workbook: Practical Ways to Implement SRE. O'Reilly Media.

[12] Burns, B., Beda, J., and Hightower, K. (2019). Kubernetes: Up and Running (2nd ed.). O'Reilly Media.

[13] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. ACM Queue, 14(1), 70–93.

[14] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4).

[15] Chambers, B., and Zaharia, M. (2018). Spark: The Definitive Guide. O'Reilly Media.

[16] Chen, M., Mao, S., and Liu, Y. (2014). Big data: A survey. Mobile Networks and Applications, 19(2), 171–209.

[17] Davoudian, A., and Liu, M. (2020). Big data systems: A software engineering perspective. ACM Computing Surveys, 53(5), 1–39.

[18] Dean, J., and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107–113.

[19] Forsgren, N., Humble, J., and Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps. IT Revolution Press.

[20] Fowler, M. (2014). Microservices: A definition of this new architectural term. martinfowler.com.

[21] Fowler, M., and Lewis, J. (2014). Microservices. martinfowler.com.

[22] Helm Project. (2019). Helm: The package manager for Kubernetes. https://helm.sh/

[23] Hightower, K., Burns, B., and Beda, J. (2017). Kubernetes Up and Running: Dive into the Future of Infrastructure. O'Reilly Media.

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

[25] Ibryam, B., and Huss, R. (2019). Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications. O'Reilly Media.

[26] Islam, M., Huang, A. K., Battisha, M., Chiang, M., Srinivasan, S., Peters, C., Neumann, A., and Abdelnur, A. (2012). Oozie: Towards a scalable workflow management system for Hadoop. Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, 1–10.

[27] Karau, H., and Warren, R. (2017). High Performance Spark. O'Reilly Media.

[28] Kim, G., Humble, J., Debois, P., and Willis, J. (2016). The DevOps Handbook. IT Revolution Press.

[29] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

[30] Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. Proceedings of the NetDB Workshop.

[31] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 558–565.

[32] Limoncelli, T. A. (2018). GitOps: A path to more self-service IT. ACM Queue, 16(3), 13–26.

[33] Limoncelli, T. A., Hogan, C. J., and Chalup, S. R. (2014). The Practice of System and Network Administration (3rd ed.). Addison-Wesley.

[34] Marz, N., and Warren, J. (2015). Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning.

[35] Microsoft. (2020). Azure Kubernetes Service documentation. https://docs.microsoft.com/azure/aks/

[36] Microsoft. (2020). Azure Key Vault and managed identities documentation. https://docs.microsoft.com/azure/key-vault/

[37] Morris, K. (2016). Infrastructure as Code: Managing Servers in the Cloud. O'Reilly Media.

[38] Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media.

[39] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

[40] Pahl, C., Brogi, A., Soldani, J., and Jamshidi, P. (2019). Cloud container technologies: A state-of-the-art review. IEEE Transactions on Cloud Computing, 7(3), 677–692.

[41] Polyzotis, N., Roy, S., Whang, S. E., and Zinkevich, M. (2018). Data lifecycle challenges in production machine learning: A survey. ACM SIGMOD Record, 47(2), 17–28.

[42] Russom, P. (2017). Data lakes: Purposes, practices, patterns, and platforms. TDWI Best Practices Report.

[43] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J. F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. Advances in Neural Information Processing Systems, 28, 2503–2511.

[44] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. USENIX Annual Technical Conference, 205–218.

[45] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 1–10.

[46] Singh, K., Behzad, B., and Ross, R. (2020). KEDA: Kubernetes-based event-driven autoscaling. Cloud Native Computing Foundation project documentation.

[47] Sridharan, C. (2018). Distributed Systems Observability. O'Reilly Media.

[48] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: A warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2(2), 1626–1629.

[49] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. (2014). Storm @Twitter. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 147–156.

[50] Turnbull, J. (2014). The Docker Book: Containerization Is the New Virtualization. James Turnbull.

[51] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache Hadoop YARN: Yet another resource negotiator. Proceedings of the 4th Annual Symposium on Cloud Computing, 1–16.

[52] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at Google with Borg. Proceedings of the 10th European Conference on Computer Systems, 1–17.

[53] Walls, C. (2016). Spring Boot in Action. Manning Publications.

[54] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache Spark: A unified engine for big data

processing. Communications of the ACM, 59(11), 56–65.

[55] Armbrust, M., et al. (2021). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. Proceedings of CIDR 2021.