Vrije Universiteit Amsterdam

Master Thesis

Benchmarking Infrastructure-as-Code Provisioning through Lifecycle Measurements and Dependency Graphs

Author: Tin Lai Choi (2850096)

1st supervisor: Daniele Bonetta
daily supervisor: Matthijs Jansen
2nd reader: Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 7, 2026

Abstract

Infrastructure-as-Code (IaC) is widely used to make cloud infrastructure reproducible, configurable, and automated. However, reproducible infrastructure definitions do not by themselves guarantee predictable provisioning behavior. Even when semantically similar infrastructure is provisioned from a clean scenario state, the time until resources become usable may differ across IaC tools, cloud providers, resource types, and repeated executions. Moreover, tool-reported completion does not always coincide with cloud-observed readiness, and aggregate deployment time alone does not explain how resource dependencies shape provisioning behavior.

This thesis studies IaC provisioning as a lifecycle-aware and dependency-structured process. It makes three contributions. First, it defines a dependency-graph analysis method that represents benchmark-inspired infrastructure scenarios as normalized scenario-resource graphs, enabling structural comparison and runtime annotation across tools and cloud providers. Second, it designs and implements a lifecycle-aware benchmarking framework that executes IaC provisioning scenarios under a standardized workflow, captures tool-side apply timing and cloud-side readiness observations, and produces graph, timeline, and summary artifacts for analysis. Third, it evaluates Terraform, OpenTofu, and Pulumi across AWS, Azure, and GCP using three benchmark-inspired infrastructure scenarios derived from YCSB, CloudSuite, and DeathStarBench.

The empirical evaluation covers 27 tool-provider-scenario configurations with five successful replications each. The results show that provisioning behavior is shaped more strongly by scenario and provider effects than by a globally dominant IaC tool. Apply time usually dominates deployment time-to-ready, but readiness lag remains important in selected configurations where post-apply stabilization affects practical usability. The evaluation also shows that fast configurations are not necessarily the most stable, and that resource-level readiness

behavior depends on both resource role and cloud provider. Finally, dependency graphs, runtime-dominant dependency paths, and fine-grained timelines explain why aggregate timing results differ: graph depth alone is insufficient, because compact graphs can still be slow when a provider-specific managed resource dominates timing, while larger graphs can accumulate long timing windows across several managed-service branches. Overall, the thesis demonstrates that IaC provisioning should be benchmarked through both lifecycle-aware measurement and dependency-based explanation.

Contents

1

Introduction

Cloud deployment has become a basic part of how modern digital services and computing experiments are built and operated. Web applications, data systems, and research workloads increasingly rely on cloud infrastructure to provide resources on demand, scale with changing needs, and support fast deployment and iteration (1, 2, 3). At the same time, provisioning cloud resources is no longer a simple task of launching a single virtual machine. Deployments in practice often involve a combination of networking, compute, storage, load balancing, managed services, and, in more complex settings, container platforms and application-stack components (4, 5, 6). This growing infrastructure complexity makes deployment behavior an important practical concern, because it can affect experiment turnaround time, operational cost, and the point at which provisioned resources become actually usable (4, 5, 6).

Infrastructure-as-Code (IaC) has emerged as an important way to manage this complexity. By describing desired infrastructure in source code, IaC makes deployments more reproducible, configurable, and automatable, and helps users apply software engineering practices to cloud provisioning (3, 7, 8). However, practitioners can choose from different IaC tools and different cloud providers, and these choices may lead to different provisioning behavior even for semantically similar infrastructure setups (2, 9, 10). In practice, such deployments are better understood as dependency-rich infrastructure composed of multiple interacting resources whose structure shapes ordering, parallelism, and provisioning behavior (11). This dependency-rich structure matters because deployment cannot be treated only as the moment an IaC tool reports successful completion, as resources may still require additional stabilization before they are actually ready for use (10, 12). For that reason, what is needed is a comparative benchmarking perspective that is cross-cloud, lifecycle-aware, and able to evaluate provisioning behavior in terms of deployment,

readiness, variability, and the resource-graph structure through which infrastructure is provisioned (2, 11, 12).

1.1 Problem Statement

Cloud infrastructure provisioning has evolved from simple single-resource deployment toward dependency-rich infrastructure scenarios that combine networking, compute, storage, ingress, and managed services, and in more complex cases also orchestration platforms and application-supporting components. Existing cloud benchmarking work has compared providers and configurations, and existing IaC research has established infrastructure as code as a reproducible and programmable way to define infrastructure (2, 3, 7, 8). However, prior work has not yet provided a dependency-based analysis method for IaC provisioning through which infrastructure scenarios can be represented as explicit cloud resource graphs, compared through structural characteristics, and later interpreted consistently across tools and cloud providers (2, 3, 7, 8, 11). This yields the first problem (**P1**): treating cloud provisioning as isolated resource deployment or ad hoc scenario construction limits structural comparison and explanation across tools and cloud providers; addressing this limitation requires a dependency-based analysis method that can represent resource dependencies as graphs.

Although IaC makes provisioning programmable, repeatable, and automatable, provisioning behavior cannot be understood from automation alone. For comparative evaluation, it also matters how runs are executed under a common lifecycle, when resources become actually ready after tool-reported completion, and how such behavior can be measured consistently across tools and cloud providers. Existing framework-oriented work has shown that IaC can support automated benchmark execution and metric collection, and adjacent systematic benchmarking work has shown the value of making benchmark scope, metrics, and architecture explicit (3, 12, 13). However, existing work still does not provide an operational lifecycle-aware IaC benchmarking framework that jointly captures tool-side execution, cloud-observed readiness, and dependency-graph information for comparable cross-tool and cross-cloud analysis (3, 10, 12, 13). This yields the second problem (**P2**): current IaC-based automation supports repeatable provisioning, but does not yet provide a lifecycle-aware and comparable way to measure provisioning behavior beyond tool-reported completion or to assess actual resource readiness.

As IaC adoption grows, users and experimenters increasingly need to know not only whether provisioning succeeds, but also how long provisioned environments take to be-

come usable and how stable those outcomes remain across repeated runs (7, 8, 10). Prior evaluations have reported provider-level benchmark results, cloud performance comparisons, or exploratory IaC-related measurements, and adjacent systematic benchmarking work has made methodology more explicit (2, 9, 10, 11, 12). However, these evaluations usually remain at the level of aggregate or end-to-end observations and do not connect repeated timing measurements to extracted dependency graphs, dependency-path reasoning, and fine-grained timeline visualizations (2, 10, 11, 12). This yields the third problem **(P3)**: IaC provisioning behavior cannot be adequately understood from end-to-end completion alone, but requires lifecycle-aware timing analysis together with structural dependency analysis under repeated cross-cloud comparison.

Because this thesis compares IaC provisioning across different tools and cloud providers, fairness does not mean that every provider uses identical resources. Such identical cross-cloud resources do not exist in practice. Instead, fairness means that comparisons are made under explicit and controlled rules: each configuration follows the same benchmark lifecycle, implements the same scenario-level resource roles, uses comparable resource sizes where possible, applies the same repetition and cleanup policy, and evaluates readiness through role-specific cloud-side predicates. These rules make the comparison meaningful while still allowing provider-specific implementations to differ where cloud services are inherently different.

This thesis also uses dependency paths as explanatory artifacts. A dependency path is a chain of prerequisite resources in the normalized scenario-resource graph. When timing observations are attached to the graph, the thesis identifies runtime-dominant dependency paths, meaning dependency chains that are most consistent with the latest observed tool-side finish times in a particular run. These paths should not be interpreted as proof that graph structure alone caused the observed delay. Rather, they provide an operational explanation that combines dependency structure with measured runtime behavior.

Together, these problems motivate the need for a dependency-based analysis method for IaC provisioning, a lifecycle-aware benchmarking framework that combines tool-side and cloud-side observation, and an empirical evaluation that integrates repeated end-to-end measurements with dependency-based analysis.

1.2 Research Questions

To address the problems identified above, this thesis is organized around three research questions. Each question targets one of the labelled problems and defines a distinct focus

within the overall study, from method design, to framework design, to empirical evaluation.

Research Question 1 (RQ1)

How should IaC provisioning be analyzed in terms of resource dependencies to support structural comparison and runtime annotation across tools and cloud providers?

RQ1 focuses on the design of a dependency-graph analysis method for IaC provisioning. Its purpose is to define how dependency graphs can be used, first, to compare infrastructure structure across tools and cloud providers, and second, to incorporate runtime information for later structural interpretation of provisioning behavior. This question addresses **P1**, which concerns the need to study and compare cloud provisioning as dependency-rich infrastructure scenarios rather than as isolated resources.

Research Question 2 (RQ2)

How should a lifecycle-aware benchmark framework be designed to measure IaC provisioning scenarios under explicit fairness rules and support dependency-graph analysis across tools and cloud providers?

RQ2 focuses on the design of the benchmark framework. Its purpose is to define how provisioning runs are executed under a common lifecycle, how tool-side and cloud-observed measurements are collected, and how dependency-graph information is preserved and connected to later analysis. This question addresses **P2**, which concerns the need for a structured way to measure provisioning behavior beyond tool-reported completion and to assess actual resource readiness in a comparable way across tools and cloud providers.

Research Question 3 (RQ3)

How do IaC provisioning tools compare across cloud providers and benchmark scenarios in deployment time, readiness behavior, and run-to-run variability, and what do dependency graphs, runtime-dominant dependency paths, and fine-grained provisioning timelines reveal about the structural explanations for these differences?

RQ3 focuses on the empirical evaluation of IaC provisioning behavior across tools, providers, and benchmark scenarios. Its purpose is both to compare deployment time, readiness behavior, and run-to-run variability at the end-to-end level, and to explain the observed differences through dependency graphs, runtime-dominant dependency paths, and fine-grained provisioning timelines. This question addresses **P3**, which concerns the need to understand IaC provisioning behavior not only through end-to-end completion results, but also through lifecycle-aware timing analysis together with structural dependency analysis under repeated cross-cloud comparison.

1.3 Research Methodology

This thesis uses a design-and-evaluation research methodology. The work first defines a dependency-graph analysis method for IaC provisioning, then designs and implements a lifecycle-aware benchmarking framework that instantiates this method, and finally evaluates IaC provisioning behavior across tools, cloud providers, and benchmark scenarios. The methodology is therefore organized in three parts: dependency-graph analysis method design, benchmark framework design and implementation, and empirical evaluation under controlled repeated experiments.

Dependency-graph analysis method design and framework design. To address **RQ1**, this thesis designs the dependency-graph analysis method through a sequence of method-construction steps. First, it derives method requirements from the problems identified in Section 1.1 and from the lifecycle and dependency concepts introduced in Chapter 2. Second, it defines a scenario abstraction that describes benchmark-inspired infrastructure scenarios through resource roles, cross-cloud role mappings, identity information, and observation descriptors. Third, it defines a common dependency-graph model that represents resources as nodes and prerequisite relations as directed edges. Fourth, it specifies how tool-computed graphs are extracted, cleaned, and normalized into scenario-resource graphs so that structural properties describe the infrastructure scenario rather than tool-internal artifacts. Fifth, it defines validation checks to ensure that expected scenario roles are present and that graph nodes can later be aligned with benchmark observations. Sixth, it defines runtime annotation and dependency-based reasoning procedures that attach timing observations to graph nodes and compare structural dependency-path candidates with runtime-dominant dependency paths.

To address **RQ2**, this thesis then designs and implements a lifecycle-aware benchmarking framework that instantiates the method in an executable form. The framework design follows the same stepwise logic: it derives execution and observation requirements, defines a manifest-driven configuration model, implements a standardized benchmark lifecycle, separates IaC tool drivers from cloud-provider observers, preserves identity mappings between scenario resources, IaC outputs, cloud identities, and graph nodes, and produces structured artifacts for end-to-end and dependency-based evaluation. The design is refined iteratively by checking whether the scenario abstractions, graph artifacts, timing observations, fairness rules, and output summaries remain aligned across tools, providers, and repeated runs. This iterative refinement is necessary because a design choice in one

part of the workflow, such as how a resource is named or observed, directly affects later graph normalization, runtime annotation, and evaluation.

Experimental evaluation. To address **RQ3**, this thesis adopts an experimental methodology based on repeated end-to-end executions of the benchmark scenarios under controlled configurations. The evaluation varies three main factors: IaC tool, cloud provider, and resource scenario. During each run, the framework records tool-observed apply timing, cloud-observed readiness timing, and derived metrics such as deployment time-to-ready, readiness lag, and run-to-run variability. The empirical evaluation combines two complementary analytical views within one integrated study. First, it compares end-to-end provisioning behavior across tool-provider-scenario combinations in terms of deployment time, readiness behavior, and variability. Second, it uses dependency graphs, runtime-dominant dependency-path identification, and fine-grained provisioning timelines to explain the structural explanations of the observed differences. Together, these analyses support both comparative measurement and structural dependency explanation of IaC provisioning behavior.

Reproducibility and research transparency. The methodology also emphasizes reproducibility. Dependency-graph construction rules, benchmark scenarios, and experiment configurations are defined explicitly, runs are executed through the same standardized lifecycle, and outputs are stored as structured per-run results and aggregated summaries. The framework, experiment configuration, and analysis workflow are intended to be documented and shared in reproducible form, so that the benchmark can be rerun, inspected, and extended in future work. This supports transparency of the reported results and makes both the method and the framework usable as a basis for later comparative studies of IaC provisioning behavior.

1.4 Thesis Contributions

To address the problems identified (**P1-P3**) in Section 1.1 and to answer the research questions (**RQ1-RQ3**) in Section 1.2, this thesis makes three main contributions to the study of IaC provisioning behavior. Together, these contributions provide a structured progression from method design, to framework design and implementation, to empirical evaluation across tools, cloud providers, and benchmark scenarios.

Contribution 1 (C1)

The method supports structural comparison through properties such as resource roles, dependencies, graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates.

The first contribution of this thesis is a dependency-based analysis method for IaC provisioning. The method treats IaC provisioning scenarios not as isolated resources or ad hoc example deployments, but as collections of cloud resources connected by dependency relations. It first identifies the resources and dependencies that define a provisioning scenario, then represents these relations as dependency graphs so that the scenario can be analyzed structurally across tools and cloud providers. The method supports structural comparison through properties such as resource roles, dependencies, graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates. It also supports runtime annotation by attaching provisioning observations to graph nodes, so that observed timing behavior can be interpreted in relation to resource dependencies. This contribution addresses **P1** and answers **RQ1** by defining how IaC provisioning can be analyzed in terms of resource dependencies for structural comparison and runtime annotation.

Contribution 2 (C2)

A lifecycle-aware benchmarking framework for IaC provisioning.

The second contribution of this thesis is a lifecycle-aware benchmarking framework that instantiates the analysis method in an executable and reproducible form. The framework executes benchmark-inspired provisioning scenarios under a standardized lifecycle, captures both tool-observed apply timing and cloud-observed readiness timing, and preserves the information needed for later dependency-based analysis. It combines common execution logic with tool-specific adapters, provider-specific observers, and scenario specification files that map infrastructure outputs into observable cloud resources. In this way, the framework supports comparable measurement across tools and cloud providers while going beyond tool-reported completion to assess actual resource readiness. This contribution addresses **P2** and answers **RQ2** by providing a structured and operational way to execute and measure IaC provisioning scenarios.

Contribution 3 (C3)

An empirical characterization of cross-cloud IaC provisioning behavior that integrates end-to-end benchmarking with dependency-based explanation.

The third contribution of this thesis is an empirical evaluation of IaC provisioning behavior across benchmark scenarios, cloud providers, and IaC tools. Using the method and framework developed in this work, the evaluation compares deployment time, readiness behavior, and run-to-run variability across Terraform, OpenTofu, and Pulumi on AWS, Azure, and GCP. It then explains the observed differences through dependency graphs, runtime-dominant dependency-path identification, and fine-grained provisioning timelines. Rather than reporting only aggregate end-to-end measurements, this contribution provides a structurally grounded explanation of how provisioning behavior unfolds and why cross-tool and cross-cloud differences emerge. This contribution addresses **P3** and answers **RQ3** by integrating repeated end-to-end benchmarking with dependency-based explanation.

1.5 Plagiarism Declaration

I hereby declare that this thesis is the result of my own independent work and writing. Unless explicitly cited, it does not contain material copied from other sources. Furthermore, this thesis has not been submitted for evaluation in any other course, programme, or academic context.

1.6 Thesis Structure

The thesis is organized as follows. Chapter 2 introduces the background and motivation for studying IaC provisioning as both a cloud resource lifecycle and a dependency-structured process. Chapter 3 defines the dependency-graph analysis method used to represent IaC provisioning scenarios as normalized scenario-resource graphs, compare their structural properties, and attach runtime observations to graph nodes. Chapter 4 describes the design and implementation of the lifecycle-aware benchmarking framework that executes provisioning runs, captures tool-side and cloud-side observations, and produces the artifacts required for dependency-based analysis. Chapter 5 presents the experimental setup, including the evaluated tools, cloud providers, benchmark-inspired scenarios, controlled conditions, measurement model, and the two experiments used to answer the empirical research question. Chapter 6 reports the evaluation results by first comparing deployment time-to-ready, readiness lag, and run-to-run variability across the full benchmark matrix, and then explaining selected results through dependency graphs, runtime-dominant dependency paths, and fine-grained provisioning timelines. Chapter 7 discusses related work

on IaC research, IaC-enabled cloud benchmarking, cloud-provider benchmarking, and adjacent benchmark methodologies, and positions this thesis with respect to the remaining gap. Chapter 8 concludes the thesis by summarizing the answers to the research questions, the main findings, the limitations of the study, and directions for future work.

2

Background & Motivation

This chapter provides the background needed to understand IaC provisioning as the object of study in this thesis. The chapter focuses on the concepts that motivate the design chapters that follow: IaC as a mechanism for specifying and executing cloud infrastructure, provisioning as a lifecycle that may extend beyond tool-reported completion, and resource dependencies as the structural relations through which provisioning order, parallelism, and explanation emerge. These concepts prepare for the dependency-based analysis method in Chapter 3, the lifecycle-aware benchmarking framework in Chapter 4, and the experimental setup in Chapter 5.

2.1 Infrastructure-as-Code Provisioning

The IaC provisioning lifecycle is the object of study in this thesis. Rather than treating IaC only as a programming or configuration practice, this thesis studies how IaC tools execute cloud provisioning workflows whose timing, readiness, and variability can be measured, compared, and explained. The relevant questions are how selected IaC tools create semantically aligned infrastructure scenarios, how long provisioned resources take to become usable, and how these provisioning outcomes differ across tools, cloud providers, and scenario complexity.

2.1.1 Infrastructure-as-Code and Desired-State Provisioning

IaC is commonly understood as the practice of managing and provisioning infrastructure through machine-readable definition files rather than through manual configuration or interactive administration (7, 8). In this model, infrastructure definitions describe the intended configuration of resources such as networks, compute instances, storage services,

and managed cloud services. IaC therefore turns infrastructure configuration into a software artifact that can be versioned, reviewed, reused, and executed repeatedly, supporting consistency, automation, and reproducibility in cloud provisioning (7, 8).

A central idea behind many IaC tools is desired-state provisioning. Instead of requiring users to manually perform each provisioning step, the user describes the intended infrastructure state, and the IaC tool determines the operations needed to create, update, or remove resources. Terraform, for example, describes configurations as reusable and versionable definitions for building, changing, and managing infrastructure (14). OpenTofu follows a similar model and positions itself as an open-source IaC tool that preserves Terraform-compatible workflows and configurations (15).

These properties make IaC especially relevant for cloud experiments and benchmarking. Repeated experiments require infrastructure configurations that can be recreated under controlled conditions, and IaC provides a practical mechanism for defining such configurations in a reproducible form (3). However, reproducible infrastructure definitions do not by themselves guarantee predictable provisioning timing or readiness. Even when the same configuration can be rerun, the resulting provisioning process may still differ in timing, readiness, and variability across tools and cloud providers.

2.1.2 IaC Execution Models and Thesis Scope

IaC tools differ in how users express infrastructure and how the tool turns that specification into provisioning actions. Terraform and OpenTofu use a declarative configuration model in which the user describes resources and relationships, after which the tool derives and applies the required changes (14, 15). Terraform also builds an internal dependency graph from configuration and uses this graph during planning and related operations (16). This broader dependency-oriented perspective is relevant for this thesis because provisioning timing is shaped not only by individual resources, but also by dependencies, ordering, and possible parallelism.

Pulumi follows a different model. Instead of relying only on a domain-specific configuration language, Pulumi allows infrastructure to be defined using general-purpose programming languages such as TypeScript, Python, Go, .NET, Java, and YAML (17). Despite this difference, Pulumi still provisions infrastructure through resource definitions, provider operations, and dependency relationships. For the purposes of this thesis, Pulumi is therefore included as a selected IaC provisioning tool. The thesis compares provisioning timing, readiness, and variability under semantically aligned infrastructure scenarios, not the programming models themselves.

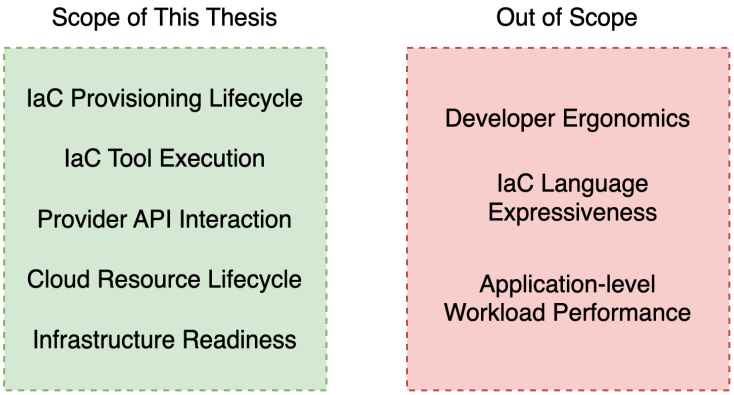

Figure 2.1: Scope of the thesis. The thesis studies the IaC provisioning lifecycle and infrastructure readiness, while excluding developer ergonomics, language expressiveness, and application-level workload performance.

This thesis further focuses on provisioning from a clean scenario state. In this thesis, a clean scenario state means that the benchmarked scenario resources do not already exist before the measured run begins. Each benchmarked run creates the resources required by the selected infrastructure scenario, observes their readiness, and then destroys the created resources after measurement. The thesis therefore does not evaluate workflows in which an IaC template attaches to, imports, updates, or extends scenario resources that already existed before the run. It also does not evaluate modifications to already deployed infrastructure. These workflows are important in operational IaC practice, but they require a different experimental setup because the initial resource state would become an additional controlled factor.

This scope motivates a dependency-based abstraction for the thesis. Different IaC tools may expose, compute, or infer dependencies differently, but their provisioned infrastructure can still be interpreted as resources connected by dependency relations. Chapter 3 formalizes this view by representing these relations as dependency graphs for structural comparison and runtime annotation.

2.1.3 Why Provisioning Timing and Readiness Matter

Provisioning timing and readiness matter because successful IaC command completion does not necessarily imply that the deployed infrastructure is already usable. A deployment may still require provider-side stabilization before resources become reachable, healthy, or ready for their intended role. This motivates the lifecycle view developed in Section 2.2.

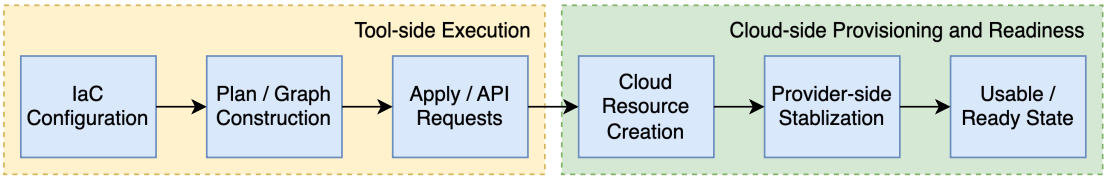

Figure 2.2: IaC provisioning as a lifecycle spanning tool-side execution and cloud-side resource creation, stabilization, and readiness.

This difference between command completion and practical usability is important in practical cloud experiments. Long provisioning times increase experiment turnaround time, while unstable provisioning times make repeated experiments less predictable. The same infrastructure scenario may also behave differently across cloud providers because managed services, networking components, and readiness conditions are implemented differently in each cloud environment (2, 9). As a result, IaC provisioning should be evaluated not only by whether a deployment succeeds, but also by how long it takes to become usable and how stable that outcome is across repeated runs.

These observations motivate the lifecycle-aware benchmarking framework developed in Chapter 4. They also motivate the dependency-based analysis method developed in Chapter 3, because timing differences are easier to interpret when resource-level observations can be related back to the dependency structure of the provisioned infrastructure.

2.2 Resource Provisioning as a Lifecycle

Common IaC workflows expose provisioning through high-level commands such as `terraform apply`, `tofu apply`, and `pulumi up`, which apply planned changes or create and update infrastructure resources (18, 19, 20). For the purpose of this thesis, however, provisioning is treated as a lifecycle rather than as a single completion event. As shown in Figure 2.2, this lifecycle spans both tool-side execution and cloud-side resource creation, stabilization, and readiness. This lifecycle view is necessary because resource creation, provider-side stabilization, and practical readiness can occur at different moments.

2.2.1 From API Request to Usable Resource

When an IaC tool provisions infrastructure, it evaluates the desired configuration, determines the changes required to reach that state, and issues provider API requests. In Terraform-like workflows, this process is supported by a dependency graph and by

providers, which act as abstractions over upstream service APIs and perform resource-specific API interactions (15, 16, 21). After the API request stage, each resource still passes through provider-specific creation and stabilization steps before it becomes usable in practice.

This lifecycle is especially important for dependency-rich infrastructure scenarios. For many cloud resource types, creation and practical readiness are separate concerns: compute instances may need to pass status checks, load balancers may need healthy targets, managed databases and caches may expose availability states, and Kubernetes nodes may report readiness through node conditions (22, 23, 24, 25, 26). Prior cloud benchmarking work emphasizes that cloud measurements must account for provider-side variability and operational conditions rather than treating cloud resources as deterministic objects (2, 9, 11). Similarly, exploratory IaC performance work has shown that deployment and management timing can differ between IaC tools and cloud services, which supports treating provisioning as an observable process rather than only as a configuration outcome (10).

For this thesis, the important implication is that an infrastructure scenario cannot be evaluated only by checking whether an IaC command terminates successfully. The relevant lifecycle extends from the beginning of tool execution to the point at which the provisioned resources are externally observed as ready. This lifecycle view directly motivates the timing model used in the benchmark framework.

2.2.2 Tool-Observed Completion and Cloud-Observed Readiness

A central distinction in this thesis is the difference between tool-observed completion and cloud-observed readiness. Tool-observed completion refers to the moment when the IaC tool reports that a resource or deployment step has completed successfully. Cloud-observed readiness refers to the later, externally observed moment when the cloud resource satisfies a readiness predicate defined by the framework, such as an instance becoming healthy, a cache becoming available, or an endpoint becoming reachable.

This distinction is important because IaC tools and cloud providers observe provisioning from different perspectives. The IaC tool observes resource operations through its provider interface, because providers are the components that translate IaC actions into resource-specific cloud API calls (21, 27). In contrast, the cloud-side observer in this thesis checks resource usability directly through provider APIs and scenario-specific readiness predicates. Prior IaC-based benchmarking work demonstrates that infrastructure definitions can support reproducible cloud benchmark execution (3). For the provisioning-focused setting

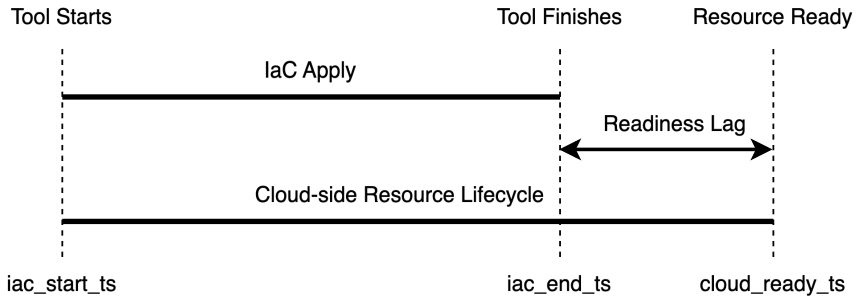

Figure 2.3: Timing distinction between tool-observed completion and cloud-observed readiness. Readiness lag captures the post-apply stabilization interval.

of this thesis, this prior work motivates separating tool-observed completion from cloud-observed readiness. The framework operationalizes this separation by recording tool-side completion and cloud-side readiness as distinct observations.

Chapter 4 operationalizes this distinction through separate tool-side and cloud-side observations. This makes it possible to measure not only when the IaC tool finishes, but also how much additional stabilization time remains before the provisioned resources are practically usable.

2.2.3 Necessity of Lifecycle-Awareness for Benchmarking

Lifecycle-awareness is necessary because a benchmark that stops at tool-reported completion can miss part of the deployment lifecycle. A resource may be reported as successfully applied while still requiring provider-side stabilization before it can support experiments or applications. For IaC provisioning, this means that different tools may reach command completion at different times, while different providers may expose different readiness delays for semantically similar resources. A lifecycle-aware benchmark can therefore distinguish tool-side execution timing from cloud-side stabilization delay and compare deployment time-to-ready, readiness lag, and run-to-run variability across tool-provider-scenario combinations.

2.3 Resource Dependencies as a Structural Lens on Provisioning

The lifecycle view introduced in Section 2.2 explains why provisioning should be measured over time. A complementary structural view is also needed, because cloud infrastructure scenarios are not only sequences of provisioning events, but also collections of resources

connected by dependencies. These dependencies determine which resources must exist before others can be created, configured, attached, or observed as usable. As a result, resource dependencies shape provisioning order, constrain possible parallelism, and provide the structural context needed to explain timing differences across tools and cloud providers.

2.3.1 Resources and Dependency Relations

A provisioning scenario consists of cloud resources and the dependency relations between them. Resources include infrastructure elements such as virtual networks, subnets, virtual machines, load balancers, managed databases, caches, Kubernetes clusters, node pools, NAT components, and container registries. A dependency relation means that one resource must exist, expose an identifier, be configured, or reach a required state before another resource can be provisioned correctly. This dependency-rich view is consistent with the broader understanding of cloud infrastructure as composed of configurable and elastic resources rather than isolated machines (1, 11).

This dependency view is also consistent with how the selected IaC tools reason about infrastructure. Terraform builds a dependency graph from configuration and uses this graph during planning and related operations (16). OpenTofu similarly builds and walks a resource dependency graph (28). Pulumi uses a different programming model, but it also tracks dependencies between resources, including automatic dependencies through resource outputs and explicit dependencies through resource options (29). These tools differ in their language models and execution mechanisms, but each must still manage resource relationships during provisioning.

2.3.2 Dependency Graphs as a Common Representation

For this thesis, the important point is not that all tools expose identical dependency information. Instead, semantically aligned infrastructure scenarios can be interpreted through comparable resource dependencies. Once resources and dependencies are made explicit, the same infrastructure scenario can be compared across tools and cloud providers, even when provider-specific services or tool-specific representations differ. This is important because cross-cloud comparison is affected by provider-specific service implementations, resource models, and readiness conditions (2, 11).

Dependency graphs provide a practical representation for this purpose. In such a graph, nodes represent resources or abstract resource roles, while directed edges represent dependency relations. The graph representation turns the implicit structure of a provisioning

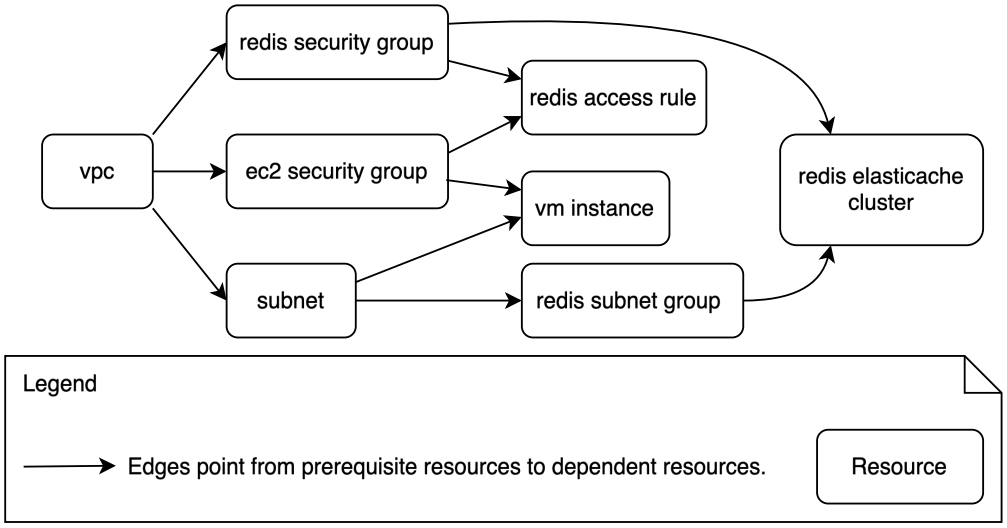

Figure 2.4: Illustrative resource dependency graph for an IaC provisioning scenario. Directed edges point from prerequisite resources to dependent resources, making provisioning dependencies explicit.

scenario into an explicit object of analysis. Instead of treating a scenario as a flat list of resources, the graph shows which resources are ordered by dependencies, where parallelism may be structurally possible, and which dependency chains may help explain provisioning timing differences. Figure 2.4 illustrates this idea with a simplified resource dependency graph.

2.3.3 Necessity of Dependency-Based Analysis

Dependency-based analysis is needed because isolated-resource reasoning is insufficient for explaining differences in IaC provisioning timing and readiness. Two infrastructure scenarios may contain similar resource types, but still differ substantially in dependency structure. Prior cloud benchmark suites and cloud workload studies also show that realistic cloud systems combine multiple interacting infrastructure and application components, rather than only isolated resources (4, 5, 6).

This matters for cross-tool and cross-cloud comparison. If only individual resources are compared, then the evaluation can show which resource took longer, but it cannot explain how resource dependencies shaped provisioning order, overlap, or bottlenecks. A dependency-graph representation makes these relationships explicit and prepares the scenario for later runtime annotation with lifecycle observations. This structural view leads directly into Chapter 3, which defines the dependency-graph analysis method used to support structural comparison and runtime annotation.

2.3.4 Implications for Framework and Evaluation

The dependency-based view developed in this section motivates the analysis method defined in Chapter 3. However, a method alone is not sufficient to study IaC provisioning in practice. The method must also be instantiated in a benchmark framework that can execute semantically aligned scenarios, collect tool-side and cloud-side observations, and attach these observations to the corresponding resource dependency graphs. This framework-level concern is addressed in Chapter 4.

The same dependency-based view also shapes the empirical evaluation. If provisioning timing is analyzed only at the end-to-end level, the evaluation can show which tool-provider-scenario combinations are faster or slower, but it cannot explain how resource dependencies, runtime-dominant dependency paths, and resource-level timelines contribute to those differences. The evaluation therefore combines repeated lifecycle measurements with dependency-based interpretation. This evaluation design is introduced in Chapter 5 and applied in Chapter 6.

3

Dependency-Graph Analysis Method

Chapter 2 motivated the need to analyze IaC provisioning not only as a lifecycle over time, but also as a dependency-structured process. This chapter addresses the first problem (**P1**) and the first research question (**RQ1**). P1 states that treating cloud provisioning as isolated resource deployment or ad hoc scenario construction limits structural comparison and explanation across tools and cloud providers. **RQ1** asks how IaC provisioning should be analyzed in terms of resource dependencies to support structural comparison and runtime annotation across tools and cloud providers.

The chapter answers **RQ1** by defining a dependency-graph analysis method. The method specifies how IaC provisioning scenarios are abstracted into scenario resource roles, how tool-computed dependency graphs are extracted and normalized into scenario-resource graphs, how structural properties such as graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates are computed, and how runtime observations are attached to graph nodes. In this way, the chapter explains not only what the method produces, but also how the method transforms IaC inputs into graph-based analysis artifacts.

The chapter focuses on the method rather than on implementation details. It first defines the scope and requirements of the method, then introduces the scenario abstraction used for cross-cloud comparison, the dependency-graph model used to represent resources and dependency edges, the graph extraction and normalization pipeline, the structural-analysis procedure, and the runtime-annotation procedure. Chapter 4 later explains how this method is instantiated in the benchmarking framework.

3.1 Method Scope and Requirements

The method addresses **P1** by making dependency structure explicit. Without an explicit representation, IaC provisioning scenarios can easily be treated as flat collections of resources or as tool-specific artifacts. Such a view is insufficient for comparing infrastructure structure across tools and cloud providers, because differences in dependency ordering, parallelizable layers, and dependency paths are hidden.

The method therefore defines a common dependency-graph representation for IaC provisioning scenarios. This representation abstracts from tool-specific graph formats while preserving the resource dependencies needed for structural comparison and runtime interpretation. In this thesis, the method is applied to provisioning scenarios derived from benchmark-inspired infrastructure roles, but the method is intended to be general enough to support additional IaC scenarios in future work.

The requirements in this section are derived from the argument developed in Chapter 2. Because provisioning is dependency-structured, the method must represent resources and dependency relations explicitly. Because the thesis compares tools and providers, the method must normalize tool-specific graphs and interpret provider-specific resources through shared scenario roles. Because provisioning is also lifecycle-aware, the method must keep enough identity information to attach runtime observations to graph nodes. These needs lead to the functional requirements (FRs) and non-functional requirements (NFRs) below.

3.1.1 Scope and Non-Scope

The scope of the method is the structural analysis of IaC provisioning through resource dependencies. It covers dependency-graph representation, graph extraction, graph normalization, scenario abstraction, structural properties, runtime annotation, and dependency-path reasoning.

The method does not evaluate developer ergonomics, programming-language expressiveness, code maintainability, or application-level workload performance. These concerns are outside the scope of the thesis. The method also does not claim that dependency graphs fully determine actual execution schedules or runtime outcomes. A dependency graph describes structural constraints and possible parallelism, while observed provisioning outcomes also depend on tool scheduling, provider APIs, cloud-side delays, retries, quotas, and readiness conditions.

3.1.2 Functional Requirements

The method must satisfy seven functional requirements (FRs).

FR1: Accept IaC templates under a defined scenario abstraction.
The method shall analyze IaC templates that conform to a method-owned scenario abstraction specification. The IaC template provides the concrete provider resources to be analyzed, while the scenario abstraction specification defines the resource roles, provider mappings, identity information, and observation descriptors needed to interpret those resources consistently across tools and cloud providers. This requirement is necessary because the method does not analyze arbitrary cloud resources in isolation; it analyzes provisioning scenarios whose resources can be linked to scenario roles, dependency-graph nodes, and later benchmark observations.

FR2: Represent IaC provisioning scenarios as dependency graphs.
The method shall represent a provisioning scenario as a directed graph in which nodes correspond to cloud resources and edges correspond to dependency relations. Terraform, OpenTofu, and Pulumi each manage resource dependencies, but expose or infer them through different tool-specific mechanisms (16, 28, 29). The graph representation provides a common structural abstraction over these tool-specific mechanisms.

FR3: Normalize tool-specific graphs into normalized scenario-resource graphs.
The method shall transform tool-computed dependency graphs into normalized scenario-resource graphs by removing or collapsing tool-internal artifacts such as provider nodes, synthetic roots, helper nodes, and non-scenario artifacts. This requirement is necessary because structural properties such as graph depth, dependency paths, and parallelizable layers should describe the infrastructure scenario, not the internal representation choices of a particular IaC tool.

FR4: Support cross-tool and cross-cloud comparison.
The method shall support comparison across IaC tools and cloud providers by mapping concrete provider resources to abstract resource roles. The goal is not to claim that provider resources are identical, but to compare semantically aligned infrastructure roles under a common graph representation. This requirement supports the comparison of scenarios whose concrete implementations differ across AWS, Azure, and GCP.

FR5: Support structural analysis.
The method shall compute structural properties from dependency graphs before runtime

observations are considered. These properties include graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates. This requirement supports comparison of scenario structure independently from runtime timing data.

FR6: Support runtime annotation.
The method shall allow runtime observations collected during benchmark runs to be attached to graph nodes. These annotations connect resource-level timing observations with dependency structure. This requirement is necessary because the evaluation aims to explain observed provisioning timelines through the resource dependency graph rather than only report aggregate end-to-end measurements.

FR7: Support dependency-path reasoning.
The method shall support both structural and runtime dependency-path reasoning. Structural dependency-path candidates are derived from graph structure alone, while runtime-dominant dependency paths are derived from timing-annotated graphs. This requirement supports the later evaluation goal of explaining which dependency chains contribute most directly to observed provisioning completion.

3.1.3 Non-Functional Requirements

The method also has four non-functional requirements (NFRs).

NFR1: Consistency.
The method should apply the same conceptual graph model across tools, providers, and scenarios. Tool-specific extraction steps may differ, but the resulting normalized graph should support comparable analysis.

NFR2: Interpretability.
The graph outputs should be understandable as cloud infrastructure structures. Nodes and edges should correspond to scenario-relevant resources, defined roles, and dependency relations rather than opaque tool-internal artifacts.

NFR3: Extensibility.
The method should allow new scenarios, tools, or cloud providers to be added by defining resource roles, mappings, and extraction rules without changing the overall analysis model.

NFR4: Separation from implementation-specific artifacts.
The method should distinguish between infrastructure resources that belong to the scenario

and tool-specific artifacts that are only present because of the internal representation of a particular IaC tool.

3.2 Method Overview

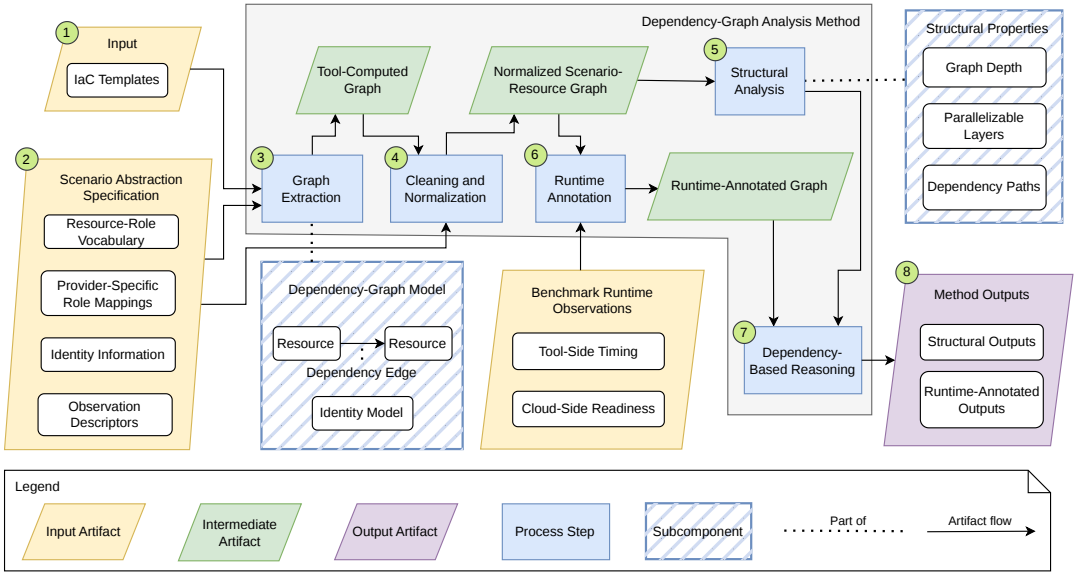

Figure 3.1: Overview of the dependency-graph analysis method, from IaC templates and method-owned scenario abstraction specification to tool-computed graphs, normalized scenario-resource graphs, structural analysis, runtime annotation, dependency-based reasoning, and method outputs.

The method overview translates the requirements from Section 3.1 into a concrete analysis pipeline. The IaC template ① and scenario abstraction specification ② provide the controlled input model required by **FR1**. Graph extraction ③, cleaning, and normalization ④ provide the common dependency representation required by **FR2**–**FR4**. Structural analysis ⑤ provides the graph-level properties required by **FR5**. Runtime annotation ⑥ provides the observation-to-node link required by **FR6**. Dependency-based reasoning ⑦ then combines the structural and runtime views required by **FR7**. The use of one normalized scenario-resource graph as the central artifact also supports the consistency, interpretability, extensibility, and separation requirements captured by the NFRs. These requirements will be discussed in more depth in later sections.

Scenario abstraction describes infrastructure scenarios at a level that is independent of a specific IaC tool or cloud provider, while still being precise enough to support graph

extraction, graph normalization, structural analysis, and runtime annotation.

The method starts from a user-provided IaC template ①. The IaC template describes the concrete provider resources to be provisioned and is the main input artifact analyzed by the method. The template is interpreted under a method-owned scenario abstraction specification ②. This specification defines the resource-role vocabulary, provider-specific role mappings, identity information, and observation descriptors needed to interpret the template as an analyzable provisioning scenario. It is therefore not treated as an ordinary per-run user input in the same sense as the IaC template. Rather, it is part of the method-level knowledge used to analyze supported scenarios.

Graph extraction ③ obtains tool-computed dependency information from the selected IaC tool. The dependency-graph model provides the conceptual basis for this step by defining resources as graph nodes, dependency relations as directed edges, and identity information as the link between IaC resources, provider resources, scenario roles, and later observations. The extracted graph is a tool-computed graph: it reflects the dependency information exposed or derived from a particular IaC tool, and may still contain provider nodes, synthetic roots, helper nodes, or other tool-internal artifacts.

Cleaning and normalization ④ transforms the tool-computed graph into a normalized scenario-resource graph. Cleaning removes or collapses tool-internal artifacts that do not represent scenario-relevant infrastructure resources. Normalization then applies scenario-role mappings and interpretable labels so that concrete provider resources can be compared through common scenario roles across tools and cloud providers.

The central intermediate artifact in the method is the *normalized scenario-resource graph*. This graph is produced after graph extraction, cleaning, and normalization. Informally, it is the dependency graph after tool-internal artifacts have been removed or collapsed and scenario-role mappings have been applied. The formal graph-artifact definition is given in Section 3.4.3.

The normalized scenario-resource graph supports two complementary analysis paths. The first path is structural analysis ⑤, which computes graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates before runtime data is considered. The structural-properties callout in Figure 3.1 summarizes these properties as details of the structural-analysis step. The second path is runtime annotation ⑥, which combines the normalized scenario-resource graph with benchmark runtime observations, including tool-side timing and cloud-side readiness observations. Runtime annotation produces a runtime-annotated graph by attaching observed timing information to the corresponding graph nodes. In the implementation evaluated in this thesis, this

runtime annotation is applied primarily through per-resource start and finish intervals parsed from the IaC tool event stream. These tool-side resource spans are later used for the all-resource timeline, runtime-attached dependency graph, and runtime-dominant dependency-path computation.

Dependency-based reasoning ⑦ combines the structural and runtime views. Structural analysis provides hypotheses about ordering constraints, possible parallelism, and structural dependency-path candidates. Runtime annotation provides evidence about when resources started and finished in the IaC tool-side timeline, while cloud-side readiness observations provide additional lifecycle context where they can be aligned with tracked scenario resources. The reasoning step compares these two forms of evidence to determine which dependency paths are structurally important, which paths are runtime-dominant, and where observed timing behavior is better explained by tool-side scheduling, provider-side processing, or readiness delays.

Finally, the method produces method outputs ⑧. These outputs include structural outputs, such as normalized scenario-resource graphs, role mappings, graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates, as well as runtime-annotated outputs, such as timing-annotated graphs, graph-linked timelines, runtime-dominant dependency paths, and runtime-dominant dependency-path metadata. These outputs are consumed by the benchmarking framework in Chapter 4 and by the experimental setup and evaluation chapters.

3.2.1 Running Example: YCSB-Inspired Data-Serving Scenario

To make the method concrete, the rest of this chapter uses a small running example based on the YCSB-inspired data-serving scenario. The example is illustrative: it shows how the method processes a compact scenario from raw IaC-derived graph information to a cleaned, normalized, and runtime-annotated graph. The scenario contains four scenario roles: `network`, `subnet`, `vm`, and `cache`. These roles represent the minimum infrastructure needed for a compact data-serving setup: a network provides the enclosing connectivity context, a subnet provides the placement scope, a virtual machine represents the client or compute-side resource, and a managed cache represents the data-serving resource.

At the tool-computed graph stage, the graph may contain more than these four scenario resources. For example, a Terraform-like graph may include provider configuration nodes, synthetic root nodes, output nodes, or helper resources in addition to concrete cloud resources such as a virtual network, subnet, virtual machine, and managed cache. In simplified form, the raw graph may contain dependencies such as:

$$\texttt{provider} \rightarrow \texttt{network} \rightarrow \texttt{subnet} \rightarrow \texttt{vm}$$

and

$$\texttt{provider} \rightarrow \texttt{network} \rightarrow \texttt{subnet} \rightarrow \texttt{cache}.$$

The exact concrete resource names differ across providers and tools. For example, the `network` role may correspond to an AWS VPC, an Azure virtual network, or a GCP VPC network. The `cache` role may likewise correspond to a provider-specific managed cache service. The method therefore does not compare raw provider resource names directly.

The cleaning step removes or collapses graph elements that do not represent scenario-relevant infrastructure resources. In the running example, provider configuration nodes, synthetic roots, and output-only nodes are removed because they are artifacts of the tool representation rather than resources in the benchmark scenario. After cleaning, the graph preserves the scenario-relevant dependency structure:

$$\texttt{network} \rightarrow \texttt{subnet} \rightarrow \texttt{vm}$$

and

$$\texttt{network} \rightarrow \texttt{subnet} \rightarrow \texttt{cache}.$$

The normalization step then maps concrete provider resources to scenario roles. For example, `aws_vpc.main`, `azurerm_virtual_network.main`, and `google_compute_network.main` can all be interpreted as the `network` role. Similarly, provider-specific subnet, virtual-machine, and managed-cache resources are mapped to `subnet`, `vm`, and `cache`. The result is a normalized scenario-resource graph that keeps the concrete dependency structure while making the nodes comparable through common scenario roles.

Runtime annotation adds timing observations to this normalized graph. Suppose one run produces tool-side resource spans in which the network finishes at 8 s, the subnet at 15 s, the virtual machine at 70 s, and the cache at 270 s after apply start. The runtime-annotated graph then contains the same dependency structure, but each node also carries observed start and finish offsets. In this example, the latest tool-side finish belongs to the cache. The runtime-dominant dependency path is therefore:

$$\texttt{network} \rightarrow \texttt{subnet} \rightarrow \texttt{cache}.$$

This does not mean that the graph structure alone proves why the run took 270 s. It means that, under the observed tool-side timing model, the cache branch best aligns with the latest finish time in this particular run. Cloud-side readiness observations, when available, are attached as additional lifecycle context and used for deployment time-to-ready and readiness-lag analysis, while the current runtime-dominant dependency-path computation uses tool-side finish offsets.

This running example will be referenced throughout the rest of the chapter. Section 3.3 explains how the four scenario roles are defined and mapped across providers. Section 3.5 explains how the raw tool-computed graph is cleaned and normalized. Section 3.6 explains how structural properties are computed before timing is considered. Section 3.7 explains how runtime observations are attached to graph nodes. Section 3.8 explains how structural dependency-path candidates and runtime-dominant dependency paths are compared.

3.3 Scenario Abstraction Method

This section defines the scenario abstraction method used in the dependency-graph analysis method. The scenario abstraction method describes how supported infrastructure scenarios are defined for analysis, while the scenario abstraction specification is the concrete method-owned artifact that records the roles, provider mappings, identity information, and observation descriptors used by the method.

The goal of scenario abstraction is not to reproduce complete application benchmarks. Instead, it defines infrastructure-oriented benchmark scenarios through resource roles, dependency expectations, cross-cloud role mappings, and observable readiness points. The abstraction therefore provides the interpretation layer between concrete IaC templates and the normalized scenario-resource graphs used later in the method.

In requirement terms, scenario abstraction is the step that makes the later graph analysis possible. It satisfies **FR1** by defining what makes a provisioning scenario analyzable, supports **FR4** through the shared role vocabulary used for cross-cloud comparison, and contributes to **NFR1** and **NFR2** by applying a consistent interpretation model that keeps graphs understandable as cloud infrastructure. Because new scenarios can be added by defining additional roles, mappings, and observation descriptors, the abstraction also supports **NFR3**.

3.3.1 Purpose and Scope of Scenario Abstraction

Scenario abstraction is needed because cross-cloud IaC benchmarking cannot rely on identical provider resources. A virtual network, subnet, load balancer, managed cache, managed database, or Kubernetes cluster may be represented by different resource types on different cloud providers. However, these resources can still play comparable roles in an infrastructure scenario. Scenario abstraction captures this role-level comparability.

The abstraction has three purposes. First, it defines which infrastructure roles belong to a scenario, such as networking, compute, routing, ingress, storage, managed services, and orchestration roles. Second, it defines how provider-specific resources are mapped to those roles, so that AWS, Azure, and GCP implementations can be compared without claiming that their concrete services are identical. Third, it defines which resources are observable during benchmark execution, so that runtime observations can later be attached to the corresponding graph nodes.

The scope of scenario abstraction is infrastructure provisioning. It preserves resource roles, dependency structure, cross-cloud interpretation, and lifecycle observability. It deliberately excludes application-level workload behavior, request throughput, business logic, and user-facing application performance. This scope is consistent with the thesis focus: the object of study is IaC provisioning behavior, not the performance of applications that may later run on the provisioned infrastructure.

Scenario abstraction also does not replace the dependency graph extracted from an IaC tool. The concrete graph remains the primary analysis object. The abstraction enriches graph nodes with scenario roles and cross-cloud interpretation, but it does not define a separate hand-written graph that overrides the tool-computed dependency graph. Structural analysis is therefore performed on the normalized scenario-resource graph, not on an abstract scenario sketch.

This scope supports **FR2** because IaC provisioning scenarios remain represented as dependency graphs, and it supports **NFR2** because those graphs remain interpretable as infrastructure structures with defined scenario resource roles.

3.3.2 Scenario Abstraction Specification

The scenario abstraction is expressed through a *scenario abstraction specification*. This specification is a method-level interpretation layer rather than an ordinary per-run user input. It defines how concrete IaC resources are interpreted as scenario resources, how

provider-specific resources are mapped to common roles, and how later runtime observations can be attached to dependency-graph nodes.

This distinction is important. The IaC template is the concrete artifact analyzed by graph extraction. The scenario abstraction specification is method-owned knowledge that defines how valid templates are interpreted. A user or framework developer may extend this specification when adding new scenarios, providers, or resource roles, but within the benchmark suite it is treated as shared method-level information rather than something that must be redefined for every run.

The scenario abstraction specification contains four kinds of information.

First, it defines the *resource-role vocabulary*. A resource role describes the function of a resource in the scenario, independent of the concrete provider-specific resource type used to implement it. Examples include `network`, `subnet`, `vm`, `cache`, `public_ingress`, `database`, `cluster`, and `node_pool`.

Second, it defines *provider-specific role mappings*. These mappings state which concrete AWS, Azure, and GCP resources implement each scenario role. For example, the role `network` may be implemented by an AWS VPC, an Azure virtual network, or a GCP VPC network. The role is the comparison unit, while the concrete resource type remains provider-specific.

Third, it defines *identity information*. This information links scenario roles, IaC addresses, provider resource types, cloud identities, and framework output keys. These identifiers are necessary because runtime observations can only be attached to dependency-graph nodes if the framework can match an observed cloud resource to the corresponding IaC resource and scenario role.

Fourth, it defines *observation descriptors*. These descriptors specify which resources are tracked by the framework and which readiness predicate is used for each tracked role. For example, a virtual machine may be considered ready when provider-specific health checks pass, while a managed cache may be considered ready when its provider-reported state becomes available.

Requirement-wise, the scenario abstraction specification connects the method inputs to the later analysis stages. It satisfies **FR1** by defining the expected scenario resources, **FR4** by recording the cross-cloud role mappings, and **FR6** by preserving the identity and observation information needed for runtime annotation. It also supports **NFR3** because new scenarios or providers can be added by extending the roles, mappings, and descriptors without changing the overall dependency-graph analysis model.

In the running YCSB-inspired example, the scenario abstraction specification defines the roles `network`, `subnet`, `vm`, and `cache`, records how each role is implemented on AWS, Azure, and GCP, and provides the identity information needed to match the provisioned VM and cache to later tool-side and cloud-side observations.

3.3.3 Scenario Abstraction Design Space

A scenario abstraction must be expressive enough to expose dependency structure that can be analyzed, but controlled enough to support repeated cross-tool and cross-cloud execution. This thesis defines the scenario abstraction design space through six dimensions (D1–D6).

D1: Benchmark-inspired infrastructure applicability.
A scenario should be grounded in an established benchmark family or cloud workload class rather than an artificial collection of unrelated resources. The goal is not to reproduce a complete application benchmark, but to preserve an infrastructure pattern that is relevant for IaC provisioning analysis. In this thesis, the YCSB-inspired scenario represents compact data-serving infrastructure, the CloudSuite-inspired scenario represents composite cloud-service infrastructure, and the DeathStarBench-inspired scenario represents broader application-stack infrastructure. This dimension is mainly tied to **NFR2**: the resulting graphs remain interpretable because their roles come from recognizable benchmark-inspired infrastructure classes rather than arbitrary resource collections.

D2: Resource-role coverage.
A scenario should define the resource roles needed to represent its infrastructure class. These roles provide the vocabulary used to interpret concrete provider resources across tools and clouds. For the requirement model, this dimension anchors **FR1** by defining the resources expected in an analyzable scenario and **FR4** by providing the shared role vocabulary used for cross-cloud interpretation. The exact role sets used in this thesis are listed in Table 3.2.

D3: Dependency-graph complexity.
A scenario should expose dependency relations that are useful for graph-based analysis. This includes dependency depth, dependency paths, parallelizable layers, and structural dependency-path candidates. Dependency-graph complexity does not simply mean that a scenario contains more resources. It means that the scenario creates opportunities to study

ordering constraints, possible parallelism, and dependency chains that may later explain provisioning behavior. This dimension provides the structural basis for **FR5** and **FR7**.

D4: Cross-cloud role coverage.
A scenario should define resource roles that can be implemented on all target cloud providers. For each scenario role, the scenario abstraction specification must identify a corresponding AWS, Azure, and GCP resource or service that satisfies the intended role-level purpose. This dimension preserves the cross-cloud comparison required by **FR4** and the consistency required by **NFR1**.

D5: Lifecycle and readiness observability.
A scenario should contain resources whose provisioning lifecycle can be observed beyond tool-reported completion. This includes resources with provider-visible readiness states or health predicates, such as virtual machines, managed services, load balancers, NAT gateways, Kubernetes clusters, and node pools. This dimension provides the observability needed for **FR6**, because benchmark observations must later be attachable to graph nodes.

D6: Practical repeatability.
A scenario must be feasible to execute repeatedly across tools, providers, and replications. This requires limiting cost, runtime, quota pressure, and operational fragility. The goal is not to build the largest possible infrastructure graph, but to define scenarios that are sufficiently realistic and structurally informative while remaining repeatable under controlled experimental conditions.

Taken together, **D1**–**D6** define the design space for scenario abstraction: the scenario must be benchmark-inspired, role-based, cross-cloud implementable, observable during provisioning, structurally informative, and practical to repeat. The next subsection positions the selected benchmark-inspired scenario instances within this design space and makes the low-to-high progression explicit through role count, readiness-observed role coverage, and expected dependency complexity.

3.3.4 Selected Scenario Instances

This thesis instantiates the scenario abstraction method through three benchmark-inspired infrastructure scenarios. The scenarios are inspired by YCSB, CloudSuite, and DeathStarBench, but they do not reproduce those benchmarks at the application-workload level. Instead, they use these benchmark families as references for three infrastructure classes with increasing dependency complexity.

Design-space dimension	YCSB-inspired	CloudSuite-inspired	DeathStarBench-inspired
D1: Benchmark-inspired infrastructure applicability	YCSB-inspired data-serving infrastructure class	CloudSuite-inspired composite cloud-service infrastructure class	DeathStarBench-inspired application-stack infrastructure class
D2: Resource-role coverage	4 roles	7 roles	10 roles
D3: Dependency-graph complexity	Lowest: compact role set and short expected dependency chains	Intermediate: additional routing, ingress, interface, and disk roles	Highest: orchestration, NAT, ingress, database, cache, and registry roles
D4: Cross-cloud role coverage	Core roles available across all providers	Wider provider-specific mapping	Broadest provider-specific mapping
D5: Lifecycle and readiness observability	VM and cache readiness	VM, ingress, and storage readiness	Ingress, NAT, Kubernetes, database, cache, and registry readiness
D6: Practical repeatability	Smallest baseline	Intermediate scenario	Feasible but highest-cost scenario

Table 3.1: Selected benchmark-inspired scenario instances positioned across the six scenario-abstraction design-space dimensions D1–D6. D1 links each scenario to its benchmark-inspired infrastructure class, while D2–D6 compare resource-role coverage, dependency-graph complexity, cross-cloud role coverage, lifecycle and readiness observability, and practical repeatability.

Table 3.1 summarizes how the three selected scenario instances occupy this design space. The table is intentionally high-level: it explains the methodological role of each scenario, while the exact resource roles are listed separately in Table 3.2.

The *YCSB-inspired data-serving scenario* represents a compact data-serving infrastructure. It provides the smallest scenario in the benchmark suite while still including both compute-side and managed-service readiness concerns. This scenario acts as the baseline for dependency-based provisioning analysis.

The *CloudSuite-inspired composite service scenario* represents a mid-sized cloud-service infrastructure. Compared with the YCSB-inspired scenario, it introduces additional roles associated with routing, public ingress, network interfaces, and persistent storage. This scenario therefore adds more dependency paths and more opportunities for structural parallelism.

The *DeathStarBench-inspired application-stack scenario* represents the largest scenario in the benchmark suite. It abstracts infrastructure for a microservice-oriented application

Scenario	Resource roles
YCSB-inspired data-serving scenario	`network`, `subnet`, `vm`, `cache`
CloudSuite-inspired composite service scenario	`network`, `subnet`, `route`, `public_ingress`, `vm`, `network_interface`, `persistent_disk`
DeathStarBench-inspired application-stack scenario	`network`, `subnet`, `route`, `nat`, `public_ingress`, `cluster`, `node_pool`, `cache`, `database`, `container_registry`

Table 3.2: Resource roles used by each benchmark-inspired scenario.

stack by including orchestration, node pools, managed services, ingress, NAT, database, and registry roles. This scenario stresses the method under the broadest role coverage, the deepest expected dependency structure, and the largest set of readiness-observable managed resources.

These scenarios are therefore selected not only because they are associated with established benchmark families, but because they instantiate distinct points in the scenario abstraction design space. Together, they allow the method to be applied across progressively more complex infrastructure graphs while remaining focused on IaC provisioning rather than application-level performance.

3.3.5 Resource Roles per Scenario

Each selected scenario is defined by a set of resource roles. A resource role describes the function of a resource in the scenario, independent of the concrete provider-specific service used to implement it. The role abstraction is necessary because cross-cloud comparison cannot rely on identical resource types. Instead, the method compares resources that serve comparable functions within the scenario.

Table 3.2 summarizes the resource-role vocabulary used by the three selected scenario instances.

The role sets follow the progression introduced above: from a compact data-serving scenario, to a composite cloud-service scenario, to a broader application-stack scenario.

These roles define what resources are expected to appear in each scenario abstraction specification. They also provide the vocabulary used during graph normalization and cross-cloud interpretation. For example, provider-specific virtual-network resources can be associated with the role `network`, while provider-specific load-balancing or gateway

resources can be associated with the role `public_ingress`. The exact provider resource types may differ, but the role labels make the scenario structure interpretable across AWS, Azure, and GCP.

The resource-role abstraction therefore connects scenario abstraction to graph analysis. It supports **FR1** because the scenario abstraction specification defines which roles are expected in an analyzable provisioning scenario. It supports **FR4** because the same role vocabulary is used for cross-tool and cross-cloud comparison. It supports **NFR2** because graph nodes can be interpreted through defined infrastructure functions rather than opaque provider-specific resource names. At the same time, roles only enrich the normalized scenario-resource graph for interpretation; they do not replace the concrete dependency graph extracted from the IaC tool.

3.3.6 Cross-Cloud Resource Mapping

Cross-cloud resource mapping is based on role equivalence rather than identical implementation. A resource in AWS, Azure, and GCP is considered comparable when it plays the same role in the scenario, even if the provider-specific resource type, configuration interface, dependency structure, and readiness condition differ.

This distinction is necessary for fairness. Requiring identical infrastructure across cloud providers would be unrealistic because providers expose different services and abstractions. However, comparing unrelated provider resources would also be unfair. The method therefore uses resource roles as the comparison unit. A provider-specific resource is included when it satisfies the intended role in the scenario abstraction and can be linked to the corresponding dependency-graph node and runtime observations.

The mapping is also explicit about its limits. Role equivalence does not mean that provider services are behaviorally identical. For example, public ingress, managed cache, managed database, and Kubernetes services may differ substantially in provisioning behavior and readiness semantics. These differences are not hidden by the abstraction. Instead, they are preserved as part of the empirical comparison: the abstraction makes the scenarios comparable, while the evaluation later shows how provider-specific implementations behave under the same role-level scenario.

The role-based mapping is the design choice that satisfies **FR4**. It gives AWS, Azure, and GCP implementations a common comparison model without claiming that their concrete services are identical. It also supports **NFR1** and **NFR2** by keeping the comparison consistent and grounded in defined scenario roles. Because new providers or resource types can be introduced by defining additional role mappings, this design also supports **NFR3**.

3.3.7 Implications for Graph Analysis and Runtime Annotation

The scenario abstraction method prepares the later stages of the dependency-graph analysis method. During graph extraction and normalization, concrete IaC resources are associated with scenario roles and non-scenario artifacts are removed or collapsed. The result is a normalized scenario-resource graph: a graph that preserves concrete resource dependency structure while making each resource interpretable through a scenario role.

During structural analysis, the scenario roles help interpret graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates. This supports **FR5** because the graph can be analyzed structurally before runtime observations are considered. During runtime annotation, the identity and observation descriptors from the scenario abstraction specification help attach tool-side and cloud-side timing observations to the correct graph nodes. This supports **FR6** because timing observations become connected to dependency structure. Once structural properties and runtime annotations are both available, the method can support **FR7** by comparing structural dependency-path candidates with runtime-dominant dependency paths.

This makes scenario abstraction a methodological step rather than a post-hoc justification of selected examples. The method first defines the design space and role vocabulary needed for dependency-based IaC provisioning analysis. The selected scenarios then instantiate this design space at three different levels of infrastructure complexity, providing the benchmark suite used by the framework and evaluation chapters.

3.4 Dependency-Graph Model

This section defines the dependency-graph model shown as a conceptual support component in Figure 3.1. The model provides the common representation used throughout the thesis for resources, dependency relations, and identity information. It is intentionally simple so that it can support comparison across IaC tools and cloud providers that expose different resource types, names, and dependency information.

Formally, a dependency graph is represented as $G = (V, E)$, where V is the set of resource nodes and E is the set of directed dependency edges. A node $v \in V$ represents a managed cloud resource that participates in the provisioning scenario. A directed edge $(u, v) \in E$ means that resource u is a prerequisite for resource v.

The method focuses on resources that belong to the scenario being analyzed. Tool-internal nodes, provider configuration nodes, synthetic roots, and helper nodes may appear in tool-computed graphs, but they are not treated as final analysis nodes unless they

correspond to scenario-relevant managed resources. Section 3.5 explains how such artifacts are removed or collapsed during normalization.

This common graph model is the main design choice for **FR2** and **NFR1**: it gives all tool-provider-scenario combinations the same conceptual representation, even when the underlying IaC tools expose different graph formats. By restricting the final analysis object to scenario-relevant resources and dependency edges, the model also supports **NFR2** by keeping the graph interpretable as cloud infrastructure rather than as a tool-internal artifact.

3.4.1 Nodes: Cloud Resources and Scenario Roles

A graph node represents a managed resource that participates in the provisioning scenario. In the normalized scenario-resource graph, nodes correspond to scenario-relevant infrastructure resources rather than to tool-internal artifacts. Examples include compute, networking, storage, managed-service, ingress, and orchestration resources, depending on the scenario being analyzed.

The method distinguishes between a concrete resource node and the scenario role associated with that node. The concrete node preserves the tool- and provider-specific resource identity needed for extraction, traceability, and runtime annotation. The scenario role provides an interpretation layer defined by the scenario abstraction in Section 3.3. These roles make it possible to compare semantically aligned infrastructure functions across tools and cloud providers without claiming that the underlying provider resources are identical.

This distinction is important because structural analysis is performed on the normalized scenario-resource graph, not on a separate abstract scenario graph. Resource roles enrich graph nodes for interpretation and cross-cloud comparison, but they do not replace the concrete resources and dependency edges extracted from the IaC tool. The detailed naming and identity information used to connect graph nodes with IaC addresses, cloud identities, and framework observations is defined in Section 3.4.4.

3.4.2 Edges: Dependency Relations

A directed edge represents a dependency relation between two resources. If the graph contains an edge (u, v), then u must exist, be configured, expose an identifier, or reach a required state before v can be provisioned correctly. For example, a virtual machine may depend on a subnet, a subnet may depend on a virtual network, and a backend service may depend on a health check or instance group.

Dependencies may be explicit or implicit. Explicit dependencies are declared directly in IaC configuration, for example through a dependency option or an explicit dependency statement. Implicit dependencies arise when one resource references outputs or identifiers produced by another resource. Terraform and OpenTofu build resource dependency graphs from configuration references and dependency declarations (16, 28). Pulumi similarly tracks dependencies through resource outputs and explicit dependency options (29).

For the purposes of this thesis, both explicit and implicit dependencies are relevant. The method does not require all tools to expose identical dependency information. Instead, it requires that the extracted graph can be normalized into a resource-level dependency representation suitable for structural comparison.

Edges complete the graph representation required by **FR2** and provide the basis for the structural analyses in **FR5** and **FR7**. Treating both explicit and implicit dependencies as directed edges also supports **NFR1**, because tool-specific dependency mechanisms are interpreted through the same graph model.

3.4.3 Graph Variants

The method distinguishes between three graph artifacts.

Tool-computed graph.
A tool-computed graph is the dependency graph exposed or derived from an IaC tool. This graph is the starting point for extraction, but it may include provider nodes, synthetic roots, helper nodes, destroy-only edges, or tool-internal artifacts.

Normalized scenario-resource graph.
A normalized scenario-resource graph is the graph produced after graph cleaning and normalization. It retains scenario-relevant managed resources and dependency edges, while removing or collapsing tool-internal artifacts such as provider configuration nodes, synthetic roots, helper nodes, and non-scenario artifacts. The graph is also enriched with scenario-role information so that concrete provider resources can be interpreted consistently across tools and cloud providers. This graph is the primary structural artifact used for dependency analysis in this thesis.

Runtime-annotated graph.
A runtime-annotated graph attaches benchmark observations to nodes in the normalized scenario-resource graph. These annotations include timing information derived from provisioning timelines, such as start offsets, finish offsets, and durations. Runtime-annotated

graphs are used for dependency-path reasoning and for explaining provisioning timing in relation to dependency structure.

The method also uses scenario abstraction specifications and resource-role mappings to interpret graph nodes and align resources across tools and cloud providers. However, these mappings are not treated as a separate abstract scenario graph in the analysis.

Figure 3.2 summarizes the relationship between these artifacts. The tool-computed graph is treated as an extraction source rather than as the final analysis object, because it may contain tool-internal nodes and edges. The normalized scenario-resource graph is the main structural artifact used by the method. The runtime-annotated graph then extends the same normalized graph with timing information, allowing structural properties and observed provisioning timelines to be analyzed together.

These graph variants make the separation required by **FR3** and **NFR4** explicit. The tool-computed graph is an extraction source, the normalized scenario-resource graph is the main analysis artifact, and the runtime-annotated graph extends that same artifact with timing observations for **FR6** and **FR7**.

3.4.4 Naming and Identity Model

The method requires a naming and identity model because graph nodes must be aligned across several domains: IaC configuration, cloud provider resources, scenario abstraction specifications, and framework observations. A resource may have one name in an IaC template, a different concrete identity in the cloud provider, and another logical name in the benchmark scenario abstraction specification. Without an explicit identity model, runtime observations cannot be reliably attached to the correct graph nodes.

Figure 3.3 shows the identifiers associated with a graph node. The *scenario role* or logical name describes the function of the resource in the benchmark scenario. The *IaC address* or tool reference identifies the resource inside the IaC tool. The *provider resource type* identifies the kind of cloud resource being created. The *cloud identity* or output key links the graph node to the concrete resource created in the provider. The *framework descriptor* defines how the benchmark framework observes the resource and checks its readiness.

This separation is important for runtime annotation. A timing observation can only be attached to a graph node if the framework can match the observation to the corresponding resource. Chapter 4 describes how this identity model is implemented through scenario specification files, IaC outputs, cloud observers, and readiness checks.

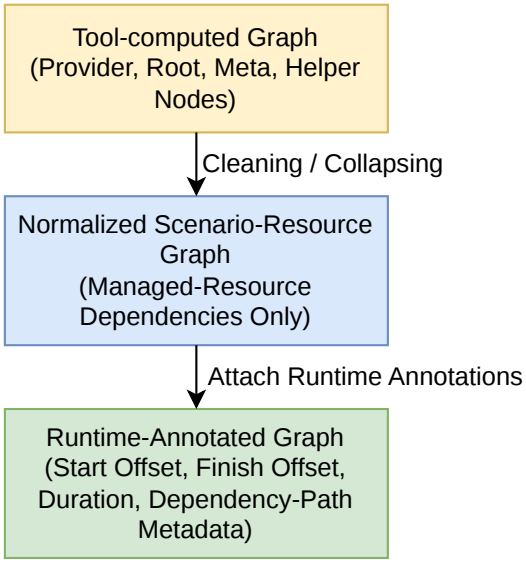

Figure 3.2: Transformation of graph artifacts used by the method. The tool-computed graph may include provider, root, meta, and helper nodes. Cleaning and collapsing produces the normalized scenario-resource graph used for structural analysis. Runtime annotations then attach start offsets, finish offsets, durations, and dependency-path information to the normalized scenario-resource graph.

The identity model is the link between scenario inputs and runtime annotation. It supports **FR1** by requiring enough input information to relate roles, IaC resources, provider resources, and framework observations, and it supports **FR6** by making those observations attachable to the correct graph nodes. Because additional identifiers can be introduced for new tools or providers, the same model also supports **NFR3**.

3.5 Graph Extraction and Normalization

This section corresponds to graph extraction ③ and cleaning and normalization ④ in Figure 3.1. It defines how tool-computed dependency information is converted into the normalized scenario-resource graph used by later analysis stages. Implementation details, including scripts and framework components, are described in Chapter 4.

The pipeline consists of four steps: extracting the tool-computed graph, cleaning tool-specific artifacts, normalizing resource names and roles, and validating graph completeness.

Stage	Main artifact	Typical representation	Purpose in the method
Input	IaC template and scenario abstraction specification	HCL files, Pulumi project files, YAML/JSON specification files	Defines concrete resources and method-owned interpretation metadata.
Extraction	Tool-computed graph	DOT, JSON, preview, plan, or equivalent tool output	Captures dependency information exposed or derived from the selected IaC tool.
Cleaning	Cleaned resource graph	Intermediate graph representation	Removes provider nodes, roots, helper nodes, and non-scenario artifacts.
Normalization	Normalized scenario-resource graph	Normalized graph JSON/SVG	Preserves scenario resources and dependency edges with role labels.
Runtime annotation	Runtime-annotated graph	Annotated graph JSON/SVG and timeline records	Attaches start offsets, finish offsets, durations, and, where available, readiness context to graph nodes.
Reasoning output	Runtime-dominant path and timeline artifacts	Tables, charts, graph-linked summaries	Supports structural comparison and runtime explanation in the evaluation.

Table 3.3: Data-artifact transformation from IaC input to dependency-based method outputs.

Together, these steps convert tool-specific dependency information into the common analysis representation defined in Section 3.4. This pipeline is the concrete design response to **FR3**: it converts tool-specific dependency information into the common analysis representation. At the same time, it supports **NFR1** by applying a consistent output model and **NFR4** by removing artifacts that belong to tool implementations rather than to the infrastructure scenario.

3.5.1 Extracting Tool-Computed Graphs

The first step is to obtain dependency information from each IaC tool. Terraform and OpenTofu expose dependency information through graph-generation functionality and use resource graphs internally during planning and operation (16, 28). Pulumi exposes dependency information differently because it uses a programming-language-based resource model, but dependencies still arise from resource outputs and explicit dependency declarations (29). This step produces the tool-computed graph artifact shown after graph

Graph Node Identity Record
Scenario-Level: Scenario Role / Logical Resource Name
IaC/Tool-Level: IaC Address / Tool Reference
Provider-Level: Provider Resource Type; Cloud Identity / Output Key
Observation-Level: Framework Descriptor / Readiness Predicate

Figure 3.3: Naming and identity model for aligning dependency graphs with benchmark observations. A graph node can be associated with a scenario role, an IaC address or tool reference, a provider resource type, a cloud identity or output key, and a framework descriptor that defines how the resource is observed and checked for readiness.

extraction ③ in Figure 3.1.

Because the tools expose dependency information in different formats, extraction is tool-specific. However, the method does not compare the raw tool outputs directly. Instead, tool-specific extraction produces an intermediate graph that is later normalized into the common graph model defined in Section 3.4, preserving **FR3** while allowing tool-specific extraction differences under **NFR1**.

3.5.2 Cleaning Tool-Specific Artifacts

Tool-computed graphs may contain nodes and edges that are useful for the tool but not meaningful for the thesis analysis. Examples include provider configuration nodes, synthetic root nodes, internal module nodes, helper nodes, and destroy-specific artifacts. If these nodes are kept, graph depth, dependency paths, and parallelizable layers may reflect tool internals rather than the structure of the infrastructure scenario.

The cleaning step therefore removes graph elements that do not correspond to managed resources or relevant configuration elements in the provisioning scenario. The remaining graph preserves the dependencies between scenario-relevant resource nodes. Together with the subsequent role-normalization step, this produces the normalized scenario-resource graph artifact after cleaning and normalization ④.

Cleaning is the main mechanism through which **FR3** and **NFR4** are enforced. It ensures that structural analysis is performed on scenario resources rather than provider nodes, synthetic roots, helper nodes, or other tool-specific artifacts, which also improves the interpretability required by **NFR2**.

3.5.3 Normalizing Resource Names and Roles

This step maps tool-specific and provider-specific resource names to normalized labels and scenario roles. Normalization is necessary because the same scenario role may be implemented through different concrete resource types across cloud providers. The role vocabulary defined in Section 3.3.5 provides the reference model for this mapping.

Normalization does not replace the normalized scenario-resource graph with a separate abstract graph. Instead, it enriches the graph after artifact removal with interpretable names and role information. At the concrete level, resources retain their IaC addresses and provider resource types so that the graph can be traced back to the implementation. At the scenario level, resources are associated with roles so that the same infrastructure function can be discussed consistently across tools and providers.

This normalization step provides the role mapping required by **FR4**. It also supports **NFR2** by giving graph nodes interpretable labels and **NFR3** by allowing new providers or scenarios to be added through additional mappings rather than changes to the graph model.

3.5.4 Validating Graph Completeness

The final step in the extraction and normalization pipeline validates the normalized graph against the scenario abstraction specification. Validation checks whether expected resource roles are present, whether resource names can be matched to known scenario descriptors, and whether runtime observations can later be attached to the corresponding graph nodes.

This step prevents analysis based on incomplete or incorrectly aligned graphs. For example, if a resource is observed during runtime but cannot be matched to a graph node, then runtime annotation would be incomplete. Conversely, if a graph contains a node that is not part of the scenario abstraction specification, the node may represent a tool artifact or an untracked resource. The validation step therefore protects the interpretability of the later analysis.

Validation protects the assumptions behind **FR1**, **FR3**, and **FR6**: the input scenario must contain the expected roles, the normalized scenario-resource graph must be complete enough for structural analysis, and runtime observations must later be alignable with graph nodes. Without this check, the resulting analysis would no longer satisfy the interpretability expected by **NFR2**.

In the running example, validation checks that the normalized graph still contains the expected `network`, `subnet`, `vm`, and `cache` roles after tool-internal artifacts have been

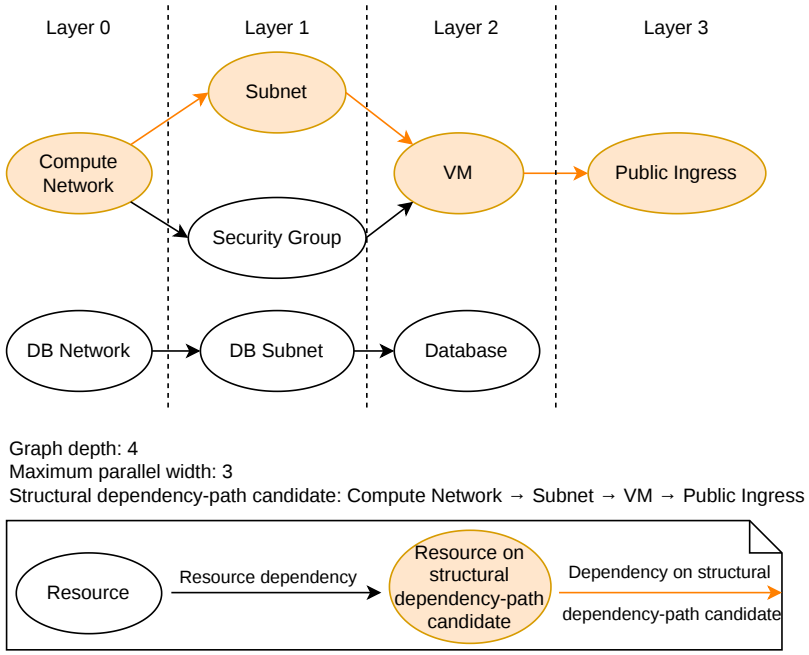

Figure 3.4: Structural properties of a dependency graph. Layering groups resources by dependency level, graph depth captures the number of dependency levels, maximum parallel width captures the largest number of resources in any layer, and dependency paths provide structural dependency-path candidates before runtime timing is considered.

removed. It also checks that these nodes can be matched to the identifiers used by the framework during runtime observation.

3.6 Structural Analysis

This section defines structural analysis ⑤ in Figure 3.1. Structural analysis examines the normalized scenario-resource graph before runtime observations are attached. The structural-properties callout in Figure 3.1 summarizes the main properties computed in this step: graph depth, parallelizable layers, and dependency paths. Figure 3.4 illustrates these concepts in more detail, including dependency layers, graph depth, maximum parallel width, dependency paths, and structural dependency-path candidates.

In this thesis, structural analysis is performed on the normalized scenario-resource graph, because this graph preserves the supporting infrastructure dependencies needed to explain provisioning structure. The analysis asks how the scenario is organized before any timing observations are considered: how deep the dependency graph is, which resources belong to

the same dependency layer, where parallel provisioning may be structurally possible, which dependency paths exist, and which paths may become structurally important during provisioning. This analysis does not claim to identify actual runtime bottlenecks. Instead, it identifies structural properties and candidate explanations that can later be tested against observed timing data.

Structural analysis is the method component that operationalizes **FR5**. It produces the structural baseline used to compare scenarios, tools, and providers before runtime data is considered, while also providing the dependency paths that later become candidates for **FR7**.

3.6.1 Graph Depth

Graph depth describes how many dependency levels are present in a provisioning graph. In a directed acyclic graph, source nodes have no predecessors and are assigned to layer 0. Each other node is assigned to one layer higher than the maximum layer of its predecessors. Let $\text{pred}(v)$ denote the set of predecessors of node v, and let $\ell(v)$ denote the dependency layer of node v. The layer of a node is defined as:

$$\ell(v) = \begin{cases} 0, & \text{if } \text{pred}(v) = \emptyset, \\ 1 + \max\limits_{u \in \text{pred}(v)} \ell(u), & \text{otherwise.} \end{cases}$$

The graph depth is then the number of dependency layers in the graph:

$$\text{depth}(G) = 1 + \max_{v \in V} \ell(v).$$

In Figure 3.4, the example graph has layers 0 through 3, so its graph depth is four.

A deeper graph indicates more sequential dependency structure. This does not automatically imply longer provisioning time, because individual resources may differ greatly in creation and readiness duration. However, graph depth is useful because it indicates how much of the scenario is structurally ordered before runtime timing observations are considered.

As one of the properties required by **FR5**, graph depth measures how sequentially ordered a provisioning graph is and helps identify long dependency chains that may later be considered as structural dependency-path candidates for **FR7**.

3.6.2 Parallelizable Layers

Parallelizable layers group resources that can be placed at the same dependency level. Resources in the same layer do not depend on one another within that layer, so they are structurally eligible for parallel provisioning. These layers can be derived using topological ordering of a directed dependency graph (30). The maximum parallel width shown in Figure 3.4 is three, because the widest layer contains three resources.

The term "parallelizable" should be interpreted carefully. A layer only describes structural potential. It does not guarantee that an IaC tool will execute all resources in the layer concurrently, nor does it guarantee that the cloud provider will process them at the same speed. Actual parallelism depends on tool scheduling, provider API processing, rate limits, retries, and resource-specific creation and readiness delays.

As another **FR5** property, parallelizable layers identify which resources are structurally eligible to be provisioned at the same dependency level. They also prepare the runtime interpretation: if resources in the same layer finish at very different times, the difference cannot be explained by dependency order alone and must be interpreted through tool-side execution timing or provider-side readiness delays.

3.6.3 Dependency Paths

A dependency path is a sequence of nodes connected by directed dependency edges. Such paths show how prerequisite resources lead to dependent resources. For example, a path may connect a network to a subnet, a virtual machine, a backend service, and a public ingress resource.

Dependency paths are useful because they reveal how provisioning structure constrains later resources. A resource near the end of many paths may depend on several earlier provisioning steps. Similarly, a long path may indicate a chain of resources that must be created in a particular order before a scenario becomes usable.

Dependency paths form the connection between **FR5** and **FR7**: they expose ordered prerequisite chains for structural analysis and provide the paths over which structural and runtime dependency-path reasoning is later performed.

3.6.4 Structural Dependency-Path Candidates

At the structural analysis stage, the method identifies structural dependency-path candidates rather than runtime-dominant dependency paths. A structural candidate is a dependency path that may matter because of its depth, role sequence, or position in the

graph. For example, a path that reaches a public ingress resource or a managed service readiness point may be structurally important because it connects many prerequisite resources to a user-visible endpoint. The highlighted path in Figure 3.4 illustrates such a structural dependency-path candidate: it is a dependency path that may be important because it connects an initial network resource to a user-visible ingress resource.

However, structural importance is not the same as runtime importance. A short dependency path containing a slow managed service may dominate the observed runtime timing, while a deeper path containing fast resources may not. For this reason, structural dependency-path candidates are treated as hypotheses for later runtime annotation, not as final explanations.

This subsection provides the structural-analysis part of **FR7**. Structural dependency-path candidates provide hypotheses about which dependency chains may matter for provisioning completion; runtime annotation later confirms, refines, or rejects those hypotheses using observed timing data.

3.6.5 Cross-Tool and Cross-Cloud Structural Comparison

The method supports structural comparison across tools and providers by comparing normalized scenario-resource graphs together with their associated resource-role mappings. Cross-tool comparison asks whether Terraform, OpenTofu, and Pulumi produce comparable dependency structures for the same scenario and provider. Cross-cloud comparison asks how the same scenario abstraction differs structurally across AWS, Azure, and GCP because of provider-specific resource models.

The comparison can consider resource-role coverage, dependency depth, layer structure, dependency paths, and structural dependency-path candidates. These comparisons help separate scenario-level structure from runtime timing outcomes. If two configurations differ structurally, then timing differences may partly reflect different dependency constraints. If two configurations are structurally similar but differ in observed timing outcomes, then the explanation may lie more in tool execution timing, provider-side API processing, or readiness delay.

3.7 Runtime Annotation

This section defines runtime annotation ⑥ in Figure 3.1. Structural analysis explains how a scenario is organized before timing observations are attached, while runtime annotation connects the normalized scenario-resource graph with observed provisioning timelines. In

Figure 3.1, runtime annotation receives two inputs: the normalized scenario-resource graph produced by ⑤ and runtime observations from benchmark runs, including tool-side timing and cloud-side readiness. In this thesis, runtime annotation means attaching these observations to the corresponding graph nodes. The method does not use runtime annotation to change execution while it is running; instead, runtime information is used after observations are available to interpret provisioning timing in relation to dependency structure.

Runtime annotation is the method component that operationalizes **FR6**. It connects the structural graph model to the lifecycle-aware measurements produced by the framework in Chapter 4, and it enables **FR7** by making dependency-path reasoning depend on both graph structure and observed timing data.

In the current framework implementation, the timing annotations used for all-resource timeline charts, runtime-attached dependency graphs, and runtime-dominant dependency-path identification are based on tool-side resource spans. Cloud-side readiness observations are preserved as lifecycle evidence and are used for deployment time-to-ready and readiness lag analysis, but they are not used as the finish metric for the runtime-dominant dependency-path algorithm.

3.7.1 Mapping Observations to Graph Nodes

Runtime annotation requires framework observations to be matched to graph nodes. This mapping uses the identity model introduced in Section 3.4.4. Depending on the available data, an observation may be matched through a logical resource name, an IaC address, an output key, a provider resource type, or a concrete cloud identity.

The result of this step is an annotated node-level view of a provisioning run. Each matched observation becomes part of the graph representation, allowing the evaluation to interpret resource-level timing in relation to dependency structure. Observations that cannot be matched are treated as incomplete annotation evidence rather than silently assigned to an incorrect node.

This mapping step is the prerequisite for **FR6**: observations can only become graph annotations if they can be aligned with the correct nodes. It also preserves **NFR2** by keeping the annotations interpretable as resource-level observations rather than disconnected timing records.

3.7.2 Timing Annotations

Timing annotations describe the observed temporal behavior associated with a graph node. In this thesis, a *tool-side resource span* refers to the per-resource start-to-finish interval parsed from the IaC tool event stream. A span consists of a start offset, a finish offset, and a duration relative to the beginning of the IaC apply operation. When available, cloud-side readiness observations can also be associated with the same node to provide additional lifecycle context.

These annotations allow the graph to be interpreted as both a structural object and a runtime object. A node is no longer only a resource in a dependency graph; it also carries observed timing data from a benchmark run. Chapter 4 defines how the framework captures the underlying tool-side and cloud-side timestamps, while this chapter defines how those observations are attached to the dependency graph.

Timing annotations are the concrete data attached under **FR6**. They also provide the start offsets, finish offsets, and durations used later by dependency-based reasoning, including runtime-dominant dependency-path identification in Section 3.8.2.

In the running example, runtime annotation turns the normalized four-role graph into a timing-annotated graph by attaching observed tool-side start and finish offsets to the `network`, `subnet`, `vm`, and `cache` nodes. This allows the later reasoning step to distinguish a structurally possible dependency path from the runtime-dominant dependency path observed in a specific run.

3.7.3 Timeline Alignment

Timeline visualizations and dependency graphs provide complementary views of the same provisioning run. A timeline shows when resources are created, completed, or observed as ready. A dependency graph shows how these resources are structurally related.

The method aligns the two views by using the same resource identity model. Once graph nodes and timeline entries refer to the same logical resources, the timeline can show temporal overlap while the graph explains dependency structure. This alignment is important for **RQ3** because end-to-end timing alone can show that one run was slower, but it cannot explain how resource dependencies contributed to that behavior.

Timeline alignment extends **FR6** from individual node annotations to graph-linked visualization. Once graph structure and timing are aligned, **FR7** can explain end-to-end timing differences through dependency paths and resource-level timing observations.

3.8 Dependency-Based Reasoning and Method Outputs

This section defines how the method combines structural analysis and runtime annotation into dependency-based reasoning outputs. The preceding sections produce two complementary forms of evidence. Structural analysis identifies graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates before runtime observations are considered. Runtime annotation attaches observed timing information to nodes in the normalized scenario-resource graph. In the current implementation, the annotations used for runtime-attached graphs and runtime-dominant dependency-path computation are tool-side resource spans, while cloud-side readiness observations are used as lifecycle evidence for deployment time-to-ready and readiness-lag interpretation. Dependency-based reasoning combines these two forms of evidence to explain observed provisioning behavior in relation to resource dependencies.

The purpose of this reasoning step is not only to report graph properties or timing measurements separately. Instead, it asks how structural candidates are supported, contradicted, or refined by runtime observations. For example, a structurally deep dependency path may not dominate runtime if its resources provision quickly, while a shorter path containing a slow managed service may explain the observed completion time. Similarly, resources in the same parallelizable layer may finish at different times, indicating that the difference cannot be explained by dependency order alone and should instead be interpreted through tool-side scheduling, provider-side processing, or readiness delays.

3.8.1 Integrating Structural and Runtime Evidence

The integration step compares structural properties with runtime annotations on the same normalized scenario-resource graph. Structural analysis provides hypotheses about possible ordering constraints and structural dependency-path candidates. Runtime annotation provides observed evidence about when resources started and finished from the IaC tool-side timeline, while cloud-side readiness observations provide additional lifecycle context for interpreting post-apply stabilization. The method combines them by asking which structural paths are actually reflected in the observed timing behavior of a run.

This integration supports **FR7**. A structural dependency-path candidate is derived from graph shape alone, while a runtime-dominant dependency path is derived from timing annotations. Comparing the two makes it possible to distinguish between paths that are structurally long and paths that are runtime-dominant. This distinction is important

because provisioning completion may be shaped by slow managed services, provider-side readiness delays, or tool-side scheduling decisions, not only by graph depth.

The result of this reasoning step is a graph-linked explanation of provisioning behavior. The method can identify which resources and dependency chains contributed most directly to observed completion, which structural candidates did not dominate runtime, and where timing differences are better explained by provider behavior or readiness delay than by dependency order alone.

3.8.2 Runtime-Dominant Dependency-Path Identification

Runtime-dominant dependency-path identification is performed on timing-annotated dependency graphs. In the current implementation, the runtime-dominant dependency path is identified from the normalized scenario-resource graph using per-node finish offsets derived from the all-resource provisioning timeline. The purpose is to identify the dependency chain that best explains the observed finishing time of a particular run.

Let $F(v)$ denote the observed finish offset of node v. The algorithm first selects the node with the latest observed finish offset:

$$v_{\text{end}} = \arg\max_{v \in V} F(v).$$

Here, arg max returns the node that maximizes the finish offset, not the finish offset value itself. The algorithm then walks backward through the graph. At each step, if the current node has multiple predecessors, the predecessor with the latest observed finish offset is selected:

$$p^*(v) = \arg\max_{u \in \text{pred}(v)} F(u).$$

If the backward traversal visits nodes $v_k, v_{k-1}, \ldots, v_0$, where $v_k = v_{\text{end}}$ and v_0 is a source node, then the runtime-dominant dependency path is written in dependency order as:

$$P_{\text{runtime}} = (v_0, v_1, \ldots, v_k).$$

The resulting sequence of nodes forms the runtime-dominant dependency path for that run. The path is runtime-specific because each predecessor is selected using measured timing data rather than graph depth alone.

This definition is intentionally different from a purely structural longest path. A structural longest path identifies a candidate path based on graph shape. A runtime-dominant

dependency path identifies the dependency chain that best aligns with the observed finishing time in a specific run. It should be interpreted as an operational explanation under the available timing model, not as proof that graph structure alone caused the observed delay. Because cloud provisioning times vary, runtime-dominant dependency paths may differ across repeated runs of the same tool-provider-scenario combination.

Runtime-dominant dependency-path identification is the runtime part of **FR7**. It turns timing-annotated dependency graphs into run-specific explanations of provisioning completion, using observed finish times rather than graph shape alone.

In the running example, if the cache node has the latest observed finish offset, the algorithm starts from `cache` and walks backward through its latest-finishing predecessors. The resulting path, `network` → `subnet` → `cache`, is the runtime-dominant dependency path for that run.

3.8.3 Structural Outputs

Structural outputs include normalized scenario-resource graphs, resource-role mappings, graph depth values, parallelizable layers, dependency paths, and structural dependency-path candidates. These outputs provide the baseline for comparing scenario structure across tools and cloud providers before runtime timing data is considered. They also provide the structural hypotheses that are later tested against runtime observations.

3.8.4 Runtime-Annotated and Reasoning Outputs

Runtime-annotated outputs include timing-annotated graphs, graph-linked resource timing summaries, runtime-dominant dependency paths, and graph-linked timeline interpretations. These outputs connect observed provisioning timing to the resource dependency structure. They support dependency-based explanation by showing not only when resources were provisioned or became ready, but also how those timings relate to dependency paths and structural dependency-path candidates.

3.8.5 Interface to Framework and Evaluation

Chapter 4 implements the method in the benchmarking framework. The framework executes provisioning runs, collects timing observations, extracts tool-computed dependency information, produces normalized scenario-resource graph artifacts, aligns graph nodes with resource timing information, and produces the artifacts required for dependency-based analysis.

Chapter 5 defines how these method outputs are used in the experimental setup. Chapter 6 then reports the evaluation results, combining end-to-end benchmarking with dependency-based explanation.

3.9 Assumptions and Limitations

The method relies on several assumptions about scenario comparability, dependency-graph extraction, structural interpretation, and runtime annotation. These assumptions define the boundary of what the method can explain. They also clarify which differences can be attributed to the method-level abstraction and which may arise from provider-specific services, tool-specific graph fidelity, or run-specific timing variability.

3.9.1 Semantic Equivalence Rather Than Identical Resources

The method relies on semantic equivalence between cross-cloud resource roles, as defined in Section 3.3.6. This makes cross-cloud comparison possible, but it also limits the interpretation of results. Observed differences may reflect provider-specific service implementations, readiness semantics, or resource models, not only differences in IaC tool execution timing.

3.9.2 Tool-Computed Graphs May Differ in Fidelity

The method assumes that useful dependency information can be extracted from each selected IaC tool, but it does not assume that all tools expose dependency graphs with the same fidelity or structure. Terraform, OpenTofu, and Pulumi differ in language model, execution model, and dependency representation (16, 28, 29). Normalization is therefore necessary, but normalization may also hide some tool-specific details.

3.9.3 Structural Dependency Is Not Actual Execution

A dependency graph represents ordering constraints and possible parallelism. It does not fully determine the actual execution schedule or cloud-side creation, stabilization, and readiness delays. Actual provisioning can be affected by tool scheduling decisions, API rate limits, provider-side queuing, retries, regional capacity, and readiness conditions.

For this reason, the method separates structural analysis from runtime annotation. Structural properties help explain possible constraints, while runtime annotations show what happened in a particular run.

3.9.4 Runtime-Dominant Dependency Paths Are Run-Specific

Runtime-dominant dependency paths depend on observed timing data. Because cloud provisioning can vary across repeated runs, the runtime-dominant dependency path for one run may not be identical to the runtime-dominant dependency path for another run. This is not a weakness of the method; rather, it reflects the variability that the thesis aims to measure and explain.

The evaluation therefore treats runtime-dominant dependency paths as run-specific explanations. When repeated runs show similar runtime-dominant dependency paths, this supports a stronger structural explanation. When repeated runs show different runtime-dominant dependency paths, this indicates that variability is distributed across multiple resources or dependency branches.

4

Benchmark Framework Design and Implementation

This chapter describes the design and implementation of the lifecycle-aware benchmarking framework. While Chapter 3 addressed **P1** and **RQ1** by defining the dependency-graph analysis method, this chapter addresses the second problem (**P2**) and the second research question (**RQ2**). **P2** states that current IaC-based automation supports repeatable provisioning, but does not yet provide a lifecycle-aware and comparable way to measure provisioning behavior beyond tool-reported completion or to assess actual resource readiness. **RQ2** asks how a lifecycle-aware benchmark framework should be designed to measure IaC provisioning scenarios under explicit fairness rules and support dependency-graph analysis across tools and cloud providers.

The chapter answers **RQ2** by explaining how the framework instantiates the method from Chapter 3 in an executable benchmark workflow. The framework loads experiment configurations, executes IaC provisioning scenarios through a standardized lifecycle, captures tool-side apply timing and cloud-side readiness observations, preserves the identity information needed to connect observations to graph nodes, extracts and normalizes dependency graphs, and produces the artifacts used by the experimental setup and evaluation chapters.

The focus is on the framework architecture and dataflow rather than on source-code structure. Implementation details are therefore described through framework responsibilities, interfaces, lifecycle stages, measurement capture, dependency-analysis implementation, output artifacts, and robustness mechanisms.

4.1 Framework Scope and Requirements

This section derives the framework requirements from the benchmark goal of this thesis. It first states the framework goal, then defines the benchmarking perspective and scope, and finally lists the functional requirements (FRs) and non-functional requirements (NFRs) that guide the framework design.

4.1.1 Goal of the Framework

The goal of the framework is to operationalize the dependency-graph analysis method defined in Chapter 3. The method requires three kinds of information: infrastructure definitions that can be represented as dependency graphs, runtime observations that can be attached to graph nodes, and output artifacts that can support end-to-end and dependency-based evaluation. The framework provides the execution and observation infrastructure needed to produce this information.

At a high level, the framework executes IaC provisioning scenarios under controlled conditions, collects tool-side and cloud-side observations, and stores the data required for dependency-based analysis. It does not only run IaC tools as automation utilities. Instead, it treats the provisioning process itself as the object of measurement. This means that the framework records when the IaC tool starts and finishes, when resources are observed by the cloud provider, when resources become ready according to scenario-specific predicates, and how these observations can be connected to the dependency graphs introduced in Chapter 3.

The framework therefore serves as the bridge between the method and the evaluation. It implements the lifecycle-aware observation model motivated in Chapter 2, instantiates the dependency-graph method from Chapter 3, provides the controlled execution structure used in Chapter 5, and produces the artifacts analyzed in Chapter 6.

4.1.2 Benchmarking Perspective and Framework Scope

Benchmarking in this thesis means repeated, controlled, and comparable evaluation of IaC provisioning timing, readiness, and variability across selected experimental factors. The goal is not to benchmark cloud-provider performance in general, nor to evaluate application-level workload performance. Instead, the system under test is the provisioning process created by the interaction between an IaC tool, a cloud provider, and an infrastructure scenario.

This scope is informed by prior cloud benchmarking work, which emphasizes explicit benchmark design, controlled measurement, and repeated experiments because cloud environments can exhibit substantial variability across providers, configurations, and time (2, 9, 11). IaC-based cloud benchmarking work has also shown that infrastructure definitions can support repeatable benchmark execution (3). However, in this thesis, IaC is not only used as an automation mechanism for running a benchmark. The IaC provisioning process itself is the object being measured.

The benchmark framework therefore has to execute semantically aligned provisioning scenarios under a common lifecycle, collect tool-side execution observations, collect cloud-side readiness observations, and preserve the dependency-graph information required for later structural analysis. This framework scope directly instantiates the method defined in Chapter 3: the dependency graph provides the structural view, while the framework provides the execution and measurement infrastructure needed to observe provisioning timelines in practice.

4.1.3 Functional Requirements

From the framework goal and benchmarking scope above, the design process yields seven functional requirements (FRs).

FR1: Load manifest-driven experiment configurations.
The framework shall load experiment manifests that specify the scenario, provider, IaC tool, template path, scenario specification file, variables, replication count, timeouts, and verification settings.

FR2: Execute IaC scenarios under a standardized lifecycle.
The framework shall execute each run through a common lifecycle consisting of initialization, apply execution, output resolution, cloud observation, and destroy.

FR3: Capture tool-side provisioning observations.
The framework shall capture IaC tool execution logs and parse tool-side timing information from Terraform, OpenTofu, and Pulumi outputs.

FR4: Capture cloud-side readiness observations.
The framework shall observe cloud resources through provider-specific observers and record cloud-side request, creation, and readiness timestamps where available.

FR5: Maintain traceability between scenario resources, IaC outputs, cloud identities, and graph nodes.
The framework shall preserve the identifiers needed to connect scenario resources, IaC tool references, provider resource identities, cloud observations, and dependency-graph nodes.

FR6: Support repeated multi-factor experiments.
The framework shall support repeated executions across IaC tools, cloud providers, and scenario levels using the same manifest-driven workflow.

FR7: Produce artifacts required for end-to-end and dependency-based evaluation.
The framework shall produce per-run artifacts, aggregated summaries, timeline artifacts, normalized scenario-resource graph artifacts, runtime-annotated graph artifacts, runtime-dominant dependency-path metadata, and graph-linked analysis artifacts needed by Chapters 5 and 6.

4.1.4 Non-Functional Requirements

The framework also has six non-functional requirements (NFRs).

NFR1: Reproducibility.
The framework should make repeated runs comparable by using manifest-driven configuration, copied templates, recorded effective variables, structured outputs, and consistent execution logic.

NFR2: Fairness.
The framework should apply the same high-level lifecycle across tools and providers while allowing provider-specific readiness checks where exact resource equivalence is impossible.

NFR3: Automation.
The framework should minimize manual intervention through command-line entry points, batch scripts, repeated-run execution, summary generation, and artifact-generation scripts.

NFR4: Observability.
The framework should preserve logs, JSON outputs, per-resource timing information, cloud observations, and quality checks so that failures and timing results can be inspected.

NFR5: Robustness.
The framework should handle failures, cleanup retries, provider-specific destroy delays, and incomplete observations in a controlled and inspectable way.

NFR6: Extensibility.
The framework should allow new tools, providers, scenarios, and observers to be added through drivers, observer implementations, scenario specification files, and registry entries.

4.2 Framework Architecture and Dataflow

This section explains the architecture and dataflow of the benchmarking framework. Figure 4.1 shows how the framework separates benchmark execution from post-run dependency analysis while preserving the artifacts needed to connect both views.

Figure 4.1 should be read as two connected paths. The first path is the experiment execution and observation path, labelled **E1**–**E8**. It starts from the manifest and scenario specification file (**E1**). Configuration loading (**E2**) reads these input artifacts and produces the effective experiment configuration. Run orchestration (**E3**) prepares the isolated run context and controls the execution of a replication. The IaC tool driver (**E4**) hides tool-specific command details behind a common interface. The provisioning lifecycle (**E5**) executes the apply workflow and captures tool-side execution evidence. Scenario resource resolution (**E6**) maps IaC outputs to tracked scenario resources and cloud identities. The cloud observer (**E7**) checks provider-side resource states and readiness predicates. Finally, the framework stores the resulting per-run artifacts (**E8**), including logs, resolved outputs, runtime observations, metrics, cleanup status, and observation-quality information.

The second path is the post-run dependency-analysis path, labelled **P1**–**P10**. It starts from the stored per-run artifacts produced by the execution path, together with the experiment configuration and runtime observations (**P1**). Dependency graph export (**P2**) obtains tool-computed dependency information for the executed configuration. Graph cleaning and normalization (**P3**) removes tool-specific artifacts and maps resources to the normalized scenario-resource representation. The resulting normalized scenario-resource graphs (**P4**) are the structural graph artifacts used for later analysis. Runtime annotation (**P5**) attaches tool-side timing observations to graph nodes, producing runtime-annotated graphs (**P6**). Dependency-based reasoning (**P7**) uses these annotated graphs to produce runtime-dominant dependency-path metadata and graph-linked analysis artifacts (**P8**). Timeline analysis (**P9**) converts runtime observations into timeline artifacts (**P10**) that show resource ordering and overlap. Together, these outputs support the dependency-based evaluation in Chapter 6.

This separation is important because framework execution and dependency analysis have different responsibilities. The execution and observation path ensures that each

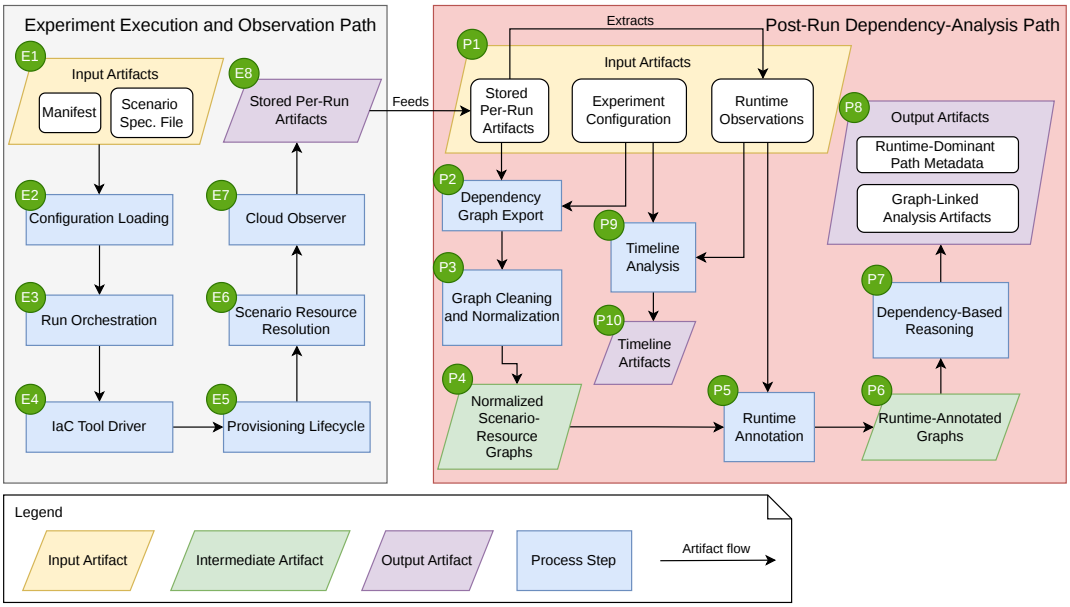

Figure 4.1: Architecture and dataflow of the benchmarking framework. The left side shows the experiment execution and observation path, labelled **E1**–**E8**. This path loads the manifest and scenario specification file, orchestrates a run, invokes the IaC tool driver and provisioning lifecycle, resolves scenario resources, observes cloud-side readiness, and stores per-run artifacts. The right side shows the post-run dependency-analysis path, labelled **P1**–**P10**. This path consumes stored per-run artifacts, experiment configuration, and runtime observations to export dependency graphs, clean and normalize them, produce normalized scenario-resource graphs, perform runtime annotation, produce runtime-annotated graphs, apply dependency-based reasoning, perform timeline analysis, and generate timeline artifacts, runtime-dominant dependency-path metadata, and graph-linked analysis artifacts. The grey region denotes experiment execution and observation, while the red region denotes post-run dependency analysis.

tool-provider-scenario combination is run under a common lifecycle and that the relevant observations are preserved. The post-run dependency-analysis path implements the graph-oriented stages of the method defined in Chapter 3: dependency graph export, graph cleaning and normalization, runtime annotation, timeline analysis, and dependency-based reasoning. Together, the two paths provide the data needed for both end-to-end benchmarking and dependency-based explanation.

4.2.1 Inputs and Outputs

The framework takes four main kinds of inputs. The first is the experiment configuration, which specifies the selected tool, provider, scenario, variables, replication settings, timeout settings, and verification settings. The second is the IaC template, which defines the provider-specific infrastructure that will be provisioned. In Figure 4.1, the IaC template is represented indirectly through the manifest because the manifest records the template path that is copied into each run context during configuration loading. The third is the scenario specification file, which is the framework-level representation of the scenario abstraction specification introduced in Chapter 3. It defines which scenario resources should be tracked, how they are resolved from IaC outputs, and which readiness predicates are used during cloud-side observation. The fourth is the execution environment, including cloud credentials, provider locations, and installed IaC tools.

The framework produces several categories of outputs. At the run level, it stores execution logs, resolved outputs, cloud observations, computed metrics, cleanup status, and observation-quality information. At the group level, it stores aggregated summaries across replications of the same experiment configuration. At the analysis level, it produces timeline artifacts, normalized scenario-resource graphs, runtime-annotated graphs, runtime-dominant dependency-path metadata, and graph-linked analysis artifacts. These outputs correspond to the method outputs described in Section 3.8: structural artifacts support dependency-graph analysis, while runtime-annotated artifacts support provisioning-timeline interpretation.

The framework therefore acts as both an executor and a recorder. It executes the provisioning workflow, but it also preserves enough structured evidence to inspect how the run unfolded, how resources were resolved, which observations were captured, and how the resulting data can be used in later evaluation.

4.2.2 Core Framework Components and Responsibility Separation

The framework is organized around a small set of conceptual components. Table 4.1 summarizes the responsibility of each component before the following subsections describe the data paths in more detail.

A configuration loader reads experiment configurations and scenario specification files. A run orchestrator creates isolated run contexts and applies the standardized lifecycle. IaC tool drivers hide tool-specific execution details behind a common interface. Scenario resolution connects IaC outputs to tracked resources. Cloud observers perform provider-specific

Component	Responsibility	Main output
Configuration loader	Reads experiment manifests, scenario specification files, variables, and execution settings.	Effective run configuration
Run orchestrator	Creates isolated run contexts and applies the standardized lifecycle.	Run context and lifecycle status
IaC tool driver	Executes the selected tool and parses tool-side events.	Tool logs, outputs, and timing spans
Scenario resolver	Maps IaC outputs to tracked scenario resources and cloud identities.	Resolved resource descriptors
Cloud observer	Checks provider-side resource states and readiness predicates.	Cloud-side timeline records
Metric and summary pipeline	Computes run-level metrics and aggregates replications.	Per-run metrics and grouped summaries
Dependency-analysis pipeline	Exports, cleans, normalizes, annotates, and reasons over dependency graphs.	Graph, timeline, and runtime-dominant dependency-path metadata

Table 4.1: Core framework components and their responsibility boundaries.

readiness observation. Metric computation derives lifecycle-aware timing measurements from tool-side and cloud-side timestamps. Summary generation aggregates repeated runs. Finally, the post-run analysis components export dependency graphs, clean and normalize graph artifacts, analyze timelines, attach runtime observations to graph nodes, perform dependency-based reasoning, and produce timeline, runtime-dominant path, and graph-linked analysis artifacts.

The purpose of this component structure is not to expose implementation details, but to separate responsibilities. Tool drivers are responsible for interacting with IaC tools; cloud observers are responsible for provider-side observation; scenario specification files are responsible for connecting abstract resource roles to concrete outputs; and graph-analysis components are responsible for turning extracted dependency information into the normalized scenario-resource graphs and runtime annotations defined in Chapter 3. This separation supports **NFR6** because new tools, providers, or scenarios can be added by extending the corresponding abstraction rather than redesigning the full framework.

4.2.3 Execution and Analysis Data Paths

The execution and observation path starts from a manifest and a scenario specification file. The framework loads the configuration, prepares the execution environment, initializes the selected IaC tool, executes the provisioning operation, parses tool-side events, resolves IaC outputs, maps those outputs to scenario resources, observes cloud-side readiness, computes metrics, performs cleanup, and stores the resulting per-run artifacts. These artifacts preserve the evidence needed for both end-to-end measurement and later dependency-based analysis.

The post-run dependency-analysis path starts from the stored per-run artifacts. The framework uses the experiment configuration to determine which tool, provider, scenario, and template were used. It extracts runtime observations from the stored run data and uses them in two ways. First, timeline analysis converts runtime observations into timeline artifacts that show the ordering and overlap of resource-level provisioning events. Second, runtime annotation attaches these observations to nodes in the normalized scenario-resource graph.

In parallel, the dependency-analysis path exports tool-computed dependency information from the selected IaC tool. The exported dependency graph is cleaned and normalized into a normalized scenario-resource graph, following the method defined in Section 3.5. Runtime annotation then combines this graph with runtime observations to produce runtime-annotated graphs. Finally, dependency-based reasoning uses the runtime-annotated graphs to compute runtime-dominant dependency-path metadata and graph-linked analysis artifacts.

The two paths are connected through identity alignment. The execution path records scenario resources, IaC outputs, cloud identities, and timing observations. The analysis path uses these identifiers to attach observations to graph nodes. This is why the identity model from Section 3.4.4 is not merely a method-level concept; it is also a framework requirement.

4.3 Configuration and Scenario Specification Model

The framework is manifest-driven. Instead of hard-coding experiment combinations into the execution logic, each benchmark configuration is described by an experiment configuration and a scenario specification file. This design supports reproducibility because the configuration that produced a run can be stored with the run artifacts. It also supports

fairness because the same workflow can be applied across tools, providers, and scenario levels.

4.3.1 Experiment Manifests

An experiment manifest defines a complete benchmark configuration. It specifies the selected scenario, cloud provider, IaC tool, infrastructure template, scenario specification file, input variables, replication count, timeout settings, and verification settings. The manifest therefore serves as the unit of repeated execution: repeated runs of the same manifest are intended to represent repeated measurements of the same tool-provider-scenario configuration.

The manifest-driven design has two advantages. First, it makes benchmark execution explicit. A run can be traced back to the configuration that selected the provider, tool, scenario, variables, and readiness policy. Second, it separates experiment definition from framework logic. The same runner can execute many experiment configurations without changing the framework itself. This directly supports **FR1** and **NFR1**.

The framework also records effective variables for each run. This is important because a run may require generated names, unique project identifiers, or provider-specific values to avoid collisions between replications. Recording the effective variables makes the run inspectable after execution and helps explain differences between repeated runs.

4.3.2 Scenario Specification Files

Scenario specification files are the framework-level representation of the method-level scenario abstraction specification introduced in Section 3.3.2. In Chapter 3, the scenario abstraction specification is defined as method-level knowledge: it describes resource roles, provider mappings, identity information, and observation descriptors. In this chapter, a scenario specification file is the concrete framework artifact that encodes the parts of that specification needed to execute and observe a benchmark run.

Scenario specification files define which resources the framework tracks and how those resources are resolved from IaC outputs. They connect the abstract resource roles from Chapter 3 to concrete outputs produced by a particular IaC template. For each tracked resource, the specification identifies the logical resource name, the tool-side reference used for matching, the output needed to obtain the cloud identity, the kind of cloud resource, and the readiness predicate used by the cloud observer.

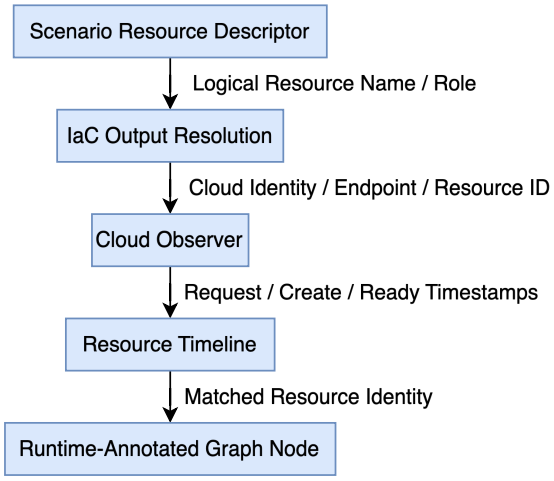

Figure 4.2: Framework-level identity alignment from scenario specification files to runtime-annotated graph nodes. Scenario resource descriptors define tracked resources and their logical roles. IaC outputs resolve these descriptors to concrete cloud identities. Cloud observers use those identities to collect request, creation, and readiness timestamps. The resulting resource timeline entries are matched back to graph nodes for runtime annotation.

This design separates infrastructure definition from observation definition. The IaC template defines how the resource is provisioned, while the scenario specification file defines how the framework should recognize and observe that resource. This separation is useful for cross-tool comparison because Terraform/OpenTofu and Pulumi templates may express the same infrastructure role differently, but the scenario specification file can still map both to the same tracked role.

Scenario specification files therefore implement the input side of the identity model described in Section 3.4.4. They define the information needed to move from scenario roles to IaC outputs, and from IaC outputs to cloud-observable resources.

4.3.3 Tracked Resource Descriptors and Identity Mapping

Figure 4.2 shows how the framework implements the naming and identity model from Section 3.4.4 by aligning scenario specification files, IaC outputs, cloud observations, resource timelines, and graph nodes. A tracked resource descriptor starts from a logical scenario resource, such as a virtual machine, cache, database, ingress component, or node pool. After provisioning, the framework resolves IaC outputs into concrete cloud identities, such as resource identifiers, names, endpoints, or cluster identifiers. Provider-specific observers then use these identities to collect request, creation, and readiness observations. Finally, these observations are aligned with dependency-graph nodes for runtime annotation.

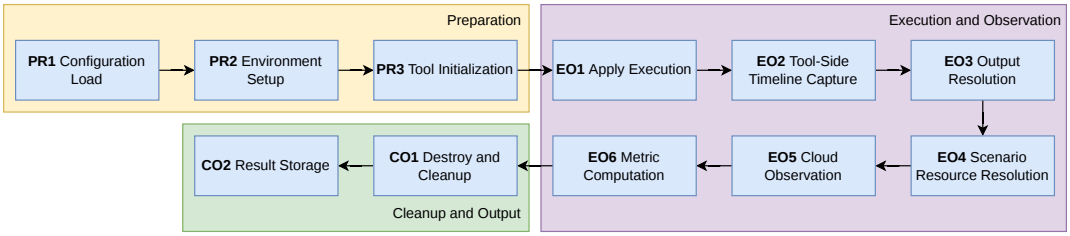

Figure 4.3: Standardized lifecycle of a benchmark run. The lifecycle is divided into preparation stages, labelled PR1–PR3; execution and observation stages, labelled EO1–EO6; and cleanup and output stages, labelled CO1–CO2. The preparation phase loads the experiment configuration (PR1), prepares the isolated execution environment (PR2), and initializes the selected IaC tool (PR3). The execution and observation phase executes the apply operation (EO1), captures the tool-side timeline (EO2), resolves IaC outputs (EO3), maps those outputs to tracked scenario resources (EO4), observes cloud-side readiness (EO5), and computes lifecycle-aware metrics (EO6). The cleanup and output phase destroys the provisioned infrastructure and performs cleanup (CO1), then stores the structured run artifacts (CO2). The same lifecycle is applied across tools, providers, and scenarios to support comparable repeated measurements.

This alignment is the framework implementation of **FR5**. Without it, tool-side logs, cloud-side observations, and dependency graphs would remain disconnected artifacts. The identity alignment allows the framework to answer questions such as which graph node a cloud readiness timestamp belongs to, which scenario role that node represents, and which tool-side event corresponds to that resource.

The alignment is also important for **NFR2** interpretability in Section 3.1.3. Instead of reporting only aggregate run duration, the framework can produce resource-level observations that are meaningful in the scenario vocabulary. This makes later evaluation possible: Chapter 6 can relate delays not only to tools or providers, but also to resource roles and dependency paths.

4.4 Standardized Benchmark Lifecycle

The preparation phase establishes the controlled context for a replication. Configuration loading (**PR1**) reads the manifest-defined experiment configuration, including the selected tool, provider, scenario, template path, variables, timeout settings, and scenario specification file. Environment setup (**PR2**) creates an isolated run context, copies the selected IaC template, records effective variables, and prepares provider-specific execution settings.

Tool initialization (**PR3**) prepares the selected IaC tool for execution, for example by initializing a Terraform-like working directory or a Pulumi project stack.

The execution and observation phase captures the evidence needed for lifecycle-aware measurement. Apply execution (**EO1**) runs the provisioning operation through the selected IaC tool driver. Tool-side timeline capture (**EO2**) records the overall apply interval and, where available, resource-level start and finish spans from the IaC tool event stream. Output resolution (**EO3**) retrieves IaC outputs after the apply operation completes. Scenario resource resolution (**EO4**) maps those outputs to tracked scenario resources and concrete cloud identities. Cloud observation (**EO5**) uses these identities together with scenario-specific readiness predicates to observe provider-side resource readiness. Metric computation (**EO6**) derives lifecycle-aware measurements such as apply duration, deployment time-to-ready, readiness lag, resource-level timing values, and observation-quality flags.

The cleanup and output phase closes the replication and preserves its evidence. Destroy and cleanup (**CO1**) removes the provisioned infrastructure and records cleanup status, retries, errors, or remaining resources where applicable. Result storage (**CO2**) stores the run configuration, tool logs, parsed timelines, resolved outputs, cloud observations, metrics, cleanup information, and status records. These artifacts are later consumed by the summary pipeline and the post-run dependency-analysis workflow described in Sections 4.2 and 4.9.

The goal of the standardized lifecycle is fairness and reproducibility. Terraform, OpenTofu, and Pulumi differ in execution model and output format, while AWS, Azure, and GCP differ in resource interfaces and readiness semantics. The framework does not eliminate these differences. Instead, it ensures that every tool-provider-scenario combination is executed and measured through the same high-level lifecycle stages. This makes the resulting measurements comparable while still allowing tool-specific event parsing, provider-specific readiness checks, and scenario-specific resource resolution.

The lifecycle is therefore a clean-state provisioning lifecycle. A run is accepted only when the scenario resources can be created from a clean run context, observed, measured, and then removed through cleanup. Existing provider-level prerequisites such as accounts, credentials, regions, quotas, and tool installations are part of the execution environment, but pre-existing scenario resources are not part of the measured benchmark target. This boundary keeps repeated runs comparable by preventing prior scenario state, imported resources, or partial existing deployments from affecting the provisioning timeline.

4.4.1 Initialization and Environment Setup

The lifecycle starts by loading the experiment configuration and preparing the run environment. The framework creates an isolated working context for the replication, copies the selected IaC template, records the effective variables, and prepares provider-specific environment settings. It also checks that the selected IaC tool is available and that the cloud identity can be queried.

This initialization stage supports reproducibility and robustness. A copied template and recorded effective configuration make the run inspectable after execution. Tool and cloud identity checks help fail early when the execution environment is not ready. Preparing a separate run context also reduces interference between replications and makes cleanup more manageable.

4.4.2 Apply Execution and Tool-Side Timeline Capture

After initialization, the framework executes the provisioning operation using the selected IaC tool. Terraform and OpenTofu follow Terraform-like apply workflows, while Pulumi uses its own stack-based update workflow (18, 19, 20). The framework treats these tool-specific mechanisms through a common driver interface so that the rest of the lifecycle can remain tool-independent.

During apply execution, the framework captures machine-readable tool output where available. This output is parsed into a tool-side timeline that records the overall apply interval and, when possible, resource-level start and finish spans. These spans provide the tool-observed view of provisioning. They do not by themselves determine cloud readiness, but they provide the reference points needed to compute apply duration and to connect tool-side events to later cloud-side observations.

This stage implements **FR3**. It ensures that the framework records not only whether the tool succeeded, but also how the tool-side provisioning process unfolded over time.

4.4.3 Output Resolution and Cloud Observation

After the apply operation completes, the framework resolves IaC outputs. These outputs provide the concrete identifiers needed to observe resources through cloud-provider APIs. The scenario specification file determines which outputs are relevant for the scenario and how they map to tracked resources.

The framework then invokes provider-specific cloud observation. The observer uses the resolved cloud identities and scenario-specific readiness predicates to record cloud-side

lifecycle timestamps. These observations may include when a resource request is visible, when a resource is observed as created, and when it satisfies the readiness condition needed for the scenario. This step implements the distinction introduced in Chapter 2 between tool-observed completion and cloud-observed readiness.

Cloud observation implements **FR4** and supports the lifecycle-aware measurement model. The key point is that an IaC tool reporting success is not treated as the final endpoint of the benchmark. The framework continues to observe the provisioned resources until the scenario-specific readiness condition is met or until the configured observation limit is reached.

4.4.4 Destroy, Cleanup, and Replication Summary

After observation and metric computation, the framework destroys the provisioned infrastructure. Cleanup is part of the benchmark lifecycle because repeated experiments must avoid accumulating resources across replications. The framework records destroy attempts, cleanup status, and any remaining resources that can be inspected after the run.

Cleanup can be affected by provider-specific delays and dependencies. For example, a cloud provider may need additional time to release network interfaces, disks, gateways, or managed-service resources before dependent resources can be deleted. The framework therefore treats cleanup as an observable part of robustness rather than assuming that a destroy command always completes immediately.

After all replications of an experiment configuration have finished, the framework produces a group-level summary. The summary records how many replications succeeded or failed and aggregates the available timing values. This supports **FR6** because repeated measurements are stored in a common format for later analysis.

4.5 IaC Tool Driver Layer

The IaC tool driver layer hides tool differences behind a common interface. The selected tools differ in configuration language, execution model, output format, and graph-export mechanisms. The framework therefore delegates tool-specific operations to drivers while keeping the benchmark lifecycle independent of the selected tool.

The driver layer is responsible for operations such as version checking, initialization, apply execution, output retrieval, destroy execution, state inspection, and timeline parsing. For dependency analysis, some drivers also provide tool-specific mechanisms for exporting or deriving dependency graphs. This design supports **NFR6** because new IaC tools can

Driver aspect	Terraform/OpenTofu	Pulumi	Framework consequence
Project setup	Template directory and initialization workflow	Pulumi project and stack workflow	Driver hides setup differences
Provisioning command	Terraform-like apply operation	Pulumi update operation	Common apply stage in the lifecycle
Output retrieval	Tool output/state-based retrieval	Stack output retrieval	Common scenario-resolution input
Event parsing	Terraform-like event and log parsing	Pulumi event-stream parsing	Tool-specific parser, normalized timing output
Graph information	Graph or plan-derived dependency output	Preview or planning-derived dependency output	Normalize after tool-specific extraction

Table 4.2: Driver differences and the framework abstractions used to keep the benchmark lifecycle comparable.

be added by implementing the same conceptual responsibilities without changing the full benchmark lifecycle.

In the driver layer, tool-specific behavior is isolated at the framework boundary. The rest of the benchmark lifecycle consumes normalized outputs: apply status, resolved outputs, tool-side timing spans, and dependency-graph inputs.

4.5.1 Common Driver Interface

The common driver interface abstracts the operations needed by the benchmark lifecycle. Each driver must be able to initialize the tool, execute provisioning, retrieve outputs, destroy resources, inspect remaining state where supported, and parse the tool output into a timeline representation.

The interface does not require all tools to expose identical information. Instead, it defines the information that the framework attempts to obtain from each tool. For example, all tools can provide an overall apply duration, but resource-level spans may differ in fidelity depending on the event stream exposed by the tool. Similarly, dependency information may be exported directly by one tool and derived from preview or plan information by another. The framework handles these differences while normalizing the resulting artifacts into the method-level representations defined in Chapter 3.

4.5.2 Terraform and OpenTofu Drivers

Terraform and OpenTofu are handled through a shared Terraform-like driver structure because their command models and graph-generation mechanisms are closely related. Both tools support initialization, apply execution, output retrieval, destroy, and state inspection through comparable workflows. Both also provide graph-oriented functionality that can be used to obtain tool-computed dependency information.

The framework captures machine-readable apply and destroy output from these tools and parses the event stream into tool-side timing spans. These spans are then used to compute the apply duration and to match resource-level tool events to scenario resources where possible. For dependency analysis, the driver obtains tool-computed graph information and passes it to the graph cleaning and normalization stage described in Section 4.8.2.

The important point is not that Terraform and OpenTofu are assumed to behave identically. Rather, the framework gives them the same lifecycle treatment and uses the same output abstractions so that their provisioning timelines and dependency graphs can be compared under controlled conditions.

4.5.3 Pulumi Driver

Pulumi requires a separate driver because it uses a different execution model based on general-purpose programming languages and project stacks. Nevertheless, the framework treats Pulumi through the same conceptual driver responsibilities: initialize the project, execute the provisioning operation, read outputs, destroy resources, capture event logs, and derive resource-level timing information where possible.

Pulumi also differs in how dependency information is obtained. Instead of relying on the same graph-generation workflow as Terraform-like tools, the framework derives graph information from Pulumi's planning or preview information. The resulting graph is then normalized into the same normalized scenario-resource graph representation used for the other tools. This allows Pulumi to be included in the dependency-based analysis without requiring the method to depend on a tool-specific graph format.

4.6 Cloud Observer Layer

The cloud observer layer captures the cloud-side part of the provisioning lifecycle. Tool drivers observe what the IaC tool reports, while cloud observers check what the cloud

Observer aspect	Common abstraction	Provider-specific implementation
Execution context	Account, project, subscription, region, or equivalent location metadata	AWS account/region, Azure subscription/location, GCP project/region/zone
Resource identity	Logical role plus resolved cloud identity	Provider-specific names, IDs, ARNs, resource-group paths, or project-scoped identifiers
Readiness predicate	Resource is considered ready or usable for the scenario	Provider-specific status checks, provisioning states, health states, or availability states
Output record	Resource timeline entry with observation quality information	Best available request, creation, and readiness timestamps for the provider and resource type

Table 4.3: Cloud observer abstraction and provider-specific responsibilities.

provider exposes about the provisioned resources. This distinction is necessary because tool-reported completion and cloud-observed readiness may occur at different times.

The observer layer is provider-specific because AWS, Azure, and GCP expose different APIs, resource states, and readiness semantics. However, the framework uses a common observer abstraction so that each provider-specific observer returns comparable resource timeline information.

The observer layer achieves comparability through a shared observation schema rather than through identical provider APIs. Provider-specific checks preserve the semantics of each cloud, while the framework stores the resulting observations in a common format.

4.6.1 Common Observer Abstraction

The common observer abstraction has two responsibilities. First, it identifies the active cloud account, project, subscription, region, or equivalent execution context so that a run can record where it executed. Second, it observes tracked resources after provisioning and returns resource-level timeline records.

A resource timeline record contains the logical resource identity, the concrete cloud identity, the cloud resource kind, and the observed request, creation, and readiness timestamps where available. These fields correspond to the lifecycle model introduced in Chapter 2 and to the runtime annotation model defined in Chapter 3.

The observer abstraction does not require all providers to expose identical timestamps for all resource types. Instead, it records the best available provider-side observations

and marks gaps through observation-quality information. This supports **NFR4** because the framework keeps the measurement evidence inspectable rather than hiding incomplete observations.

4.6.2 Provider-Specific Observers

Each cloud provider requires a provider-specific observer. The AWS observer checks AWS resource states and readiness conditions, the Azure observer checks Azure resource provisioning states and runtime states, and the GCP observer checks GCP resource status and readiness information. The exact APIs and resource states differ by provider, but each observer implements the same conceptual role: map resolved resource identities to cloud-side lifecycle observations.

Provider-specific observers are necessary for fairness. A semantically aligned resource role may be implemented differently across providers, so the framework cannot use one generic readiness check for all clouds. Instead, it uses provider-specific checks that reflect the resource model of each cloud while preserving the same high-level observation categories: request, creation, and readiness.

4.6.3 Readiness Checks

Readiness checks are scenario-specific predicates that define when a resource is considered usable for the purpose of the benchmark. Examples include a compute instance becoming healthy, a load balancer becoming active, a cache or database reporting an available state, or a managed cluster reaching a running condition. These checks are not application-level workload measurements; they are infrastructure readiness observations.

Readiness predicates are important because different resource types stabilize in different ways. A virtual machine, managed cache, database, Kubernetes cluster, and public ingress component do not expose the same readiness semantics. By making readiness checks explicit in scenario specification files, the framework avoids treating tool completion as a sufficient proxy for usability. This directly supports the lifecycle-aware benchmarking perspective of the thesis.

4.7 Measurement Capture and Metric Computation

The framework records both raw timestamps and derived metrics. Raw timestamps describe events observed by the IaC tool or the cloud provider. Derived metrics combine these timestamps into measurements used for comparison, such as apply duration, readiness lag,

cloud-side latency, and deployment time-to-ready. Table 4.4 summarizes the main timing observations and derived metrics captured by the framework.

Observation or metric	**Source**	**Unit or stored representation**
Tool start time	IaC tool driver	ISO-8601 UTC timestamp
Tool completion time	IaC tool driver	ISO-8601 UTC timestamp
Tool-side apply duration	Derived from tool start and completion times	Seconds
Tool-side resource span	Parsed IaC tool event stream	Start offset, finish offset, and duration in seconds relative to apply start
Cloud request timestamp	Cloud provider observation	ISO-8601 UTC timestamp
Cloud creation timestamp	Cloud provider resource metadata	ISO-8601 UTC timestamp
Cloud readiness timestamp	Cloud observer readiness predicate	ISO-8601 UTC timestamp
Deployment time-to-ready	Derived from tool-side and cloud-side observations	Seconds from apply start to final observed readiness
Readiness lag	Derived from tool completion time and cloud readiness time	Seconds between tool completion and observed readiness
Resource readiness duration	Derived from resource-level cloud observations	Seconds from first relevant cloud observation to readiness, when available
Run success or failure status	Run orchestrator and cleanup stage	Categorical status value
Observation quality flag	Cloud observer and metric pipeline	Categorical flag indicating complete, partial, missing, or failed observation
Replication summary statistics	Aggregated run outputs	Count, mean, standard deviation, minimum, and maximum

Table 4.4: Timing observations and derived metrics captured by the framework

The distinction between observation source and metric is important. Tool-side timing is derived from the IaC tool event stream. Cloud-side timing is derived from provider-side observation. End-to-end lifecycle metrics combine both sources. This separation allows the evaluation to distinguish between time spent in the IaC tool's execution phase and time spent waiting for cloud resources to become ready after tool completion.

4.7.1 Tool-Side Timing

Tool-side timing describes the provisioning process from the perspective of the IaC tool. The framework records when the apply operation starts, when it completes, and how long the apply operation takes. Where the tool output exposes resource-level events, the framework also records per-resource start and finish spans.

These measurements answer a limited but important question: how long did the IaC tool take to execute its provisioning operation? They do not by themselves indicate whether the resources are ready for use. However, they provide the reference interval for lifecycle decomposition. In particular, tool completion is used as the boundary between the apply phase and the post-apply readiness lag.

4.7.2 Cloud-Side Readiness Timing

Cloud-side readiness timing describes the provisioning process from the provider-side observation perspective. The framework records when resources are requested or first visible, when they are created, and when they satisfy the scenario-specific readiness predicate. These observations are collected by provider-specific cloud observers.

Cloud-side timing is necessary because tool completion and resource readiness are not always equivalent. A tool may report that a resource has been applied while the resource is still stabilizing, becoming healthy, propagating configuration, or waiting for provider-side state changes. By recording cloud-side readiness separately, the framework can measure the part of the lifecycle that would be missed by a benchmark that stops at tool completion.

4.7.3 Derived Metrics and Variability Summaries

Derived metrics combine tool-side and cloud-side observations. Tool-side apply duration measures the time between apply start and apply completion. Deployment time-to-ready measures the time from apply start to the latest relevant readiness observation. Readiness lag measures the difference between tool completion and cloud-observed readiness. Cloud creation and readiness latencies further decompose the cloud-side part of the lifecycle where timestamps are available.

For repeated runs, the framework stores the per-run values and computes group-level summary statistics such as count, mean, standard deviation, minimum, and maximum. Because the per-run values are preserved in the aggregated summary files, the evaluation in Chapter 6 can also derive median, interquartile range, and coefficient of variation

when comparing configurations. This separation keeps the framework responsible for measurement capture and basic aggregation, while Chapter 6 performs the evaluation-level statistical interpretation.

4.8 Dependency-Analysis Implementation

The dependency-analysis implementation instantiates the graph extraction, cleaning and normalization, runtime annotation, and dependency-based reasoning steps defined in Chapter 3. Its purpose is to turn tool-specific dependency information and runtime observations into the graph artifacts needed for dependency-based explanation. This section describes three implementation stages: dependency graph export, graph cleaning and normalization with scenario projection, and runtime annotation with timeline alignment and dependency-based reasoning.

4.8.1 Dependency Graph Export

The first step is to obtain dependency information from each IaC tool. Terraform and OpenTofu expose graph-oriented information through their planning and graph mechanisms, while Pulumi requires dependency information to be derived from its preview or planning artifacts. These tool-computed graphs are not compared directly. They are treated as input artifacts for the cleaning and normalization process defined in Chapter 3.

The exported graphs may contain more than scenario resources. They may include provider configuration nodes, root nodes, metadata nodes, helper nodes, or edges that are useful for tool execution but not meaningful for the thesis analysis. For this reason, export is only the first step. The framework must still transform the tool-computed graph into the normalized scenario-resource graph used for structural analysis and runtime annotation.

4.8.2 Graph Cleaning, Normalization, and Scenario Projection

Graph cleaning removes tool-specific artifacts and keeps the scenario-relevant managed resources. Normalization then maps provider-specific resource labels and tool-specific addresses to scenario-level roles and interpretable names. Scenario projection ensures that the resulting graph can be related back to the resource roles defined by the scenario abstraction specification in Chapter 3 and encoded by the framework-level scenario specification files.

This step implements the distinction between the tool-computed graph and the normalized scenario-resource graph. The normalized graph is the graph used for graph depth,

parallelizable layers, dependency paths, and dependency-path reasoning. It is also the graph that later receives runtime annotations.

The framework may also simplify the graph where appropriate, for example by removing redundant edges that do not change reachability in the resource-level dependency graph. Such simplification is used to improve interpretability, not to change the underlying scenario. The goal is to produce a graph that reflects the infrastructure dependency structure rather than the internal representation choices of a specific IaC tool.

4.8.3 Runtime Annotation, Timeline Alignment, and Dependency-Based Reasoning

Runtime annotation attaches tool-side resource timing to nodes in the normalized scenario-resource graph. In the current implementation, the graph-linked timing is derived from the all-resource IaC apply timeline, including resource start offsets, finish offsets, and durations. These annotations are used to produce timing-labelled dependency graphs and to compute runtime-dominant dependency paths.

Cloud-side request, creation, and readiness timestamps are preserved separately by the framework and are used for lifecycle decomposition, readiness-lag analysis, and resource-level interpretation. They provide important context for explaining post-apply stabilization, but the runtime-dominant dependency path in the current graph workflow is computed from IaC apply finish offsets.

As part of dependency-based reasoning, the framework also computes runtime-dominant dependency paths. As defined in Section 3.8.2, the runtime-dominant dependency path is derived from the runtime-annotated graph by starting from the node with the latest observed finish offset and walking backward through predecessors using observed finish times. When sufficient timing coverage is not available, the framework can fall back to structural dependency-path candidates, but runtime-dominant dependency paths are preferred because they explain a specific observed run.

This design choice is motivated by the granularity of the available observations. The dependency graphs are extracted from IaC tool-computed dependency information, and their nodes correspond to IaC-level resources or supporting resource nodes after cleaning and normalization. The IaC tool-side timeline therefore aligns directly with the same resource identities used in the graph. In contrast, cloud-side readiness observations are captured at the tracked resource level and are not available for every fine-grained graph node or supporting resource shown in the dependency graph. Using cloud-side readiness as the graph finish metric would therefore require coarser alignment or inference, which could make

the runtime-dominant dependency path less precise. For this reason, the current implementation uses tool-side resource spans for timeline charts, runtime-attached dependency graphs, and runtime-dominant dependency-path computation, while cloud-side readiness is retained for deployment time-to-ready, readiness lag, and lifecycle interpretation.

The dependency-analysis implementation treats graphs and timelines as aligned artifacts derived from the same run evidence, not as separate visual extras. This stage connects the two analysis views of the thesis. The timeline shows how resource events unfolded over time, while the graph explains how those resources are structurally related. The combination allows Chapter 6 to move beyond aggregate duration and explain which resource chains contributed to observed provisioning completion.

4.9 Output Artifacts and Summary Pipeline

The framework produces artifacts at three levels: per-run artifacts, aggregated summaries, and post-run analysis artifacts. These artifacts are necessary because the evaluation uses both repeated end-to-end measurements and dependency-based interpretation. The output pipeline therefore preserves raw observations, derived metrics, graph artifacts, timeline artifacts, and dependency-path information.

Table 4.5 summarizes the higher-level idea behind the output pipeline. Raw execution evidence is kept for inspection, derived metrics are kept for repeated-run comparison, and graph-linked artifacts are kept for dependency-based explanation.

4.9.1 Per-Run Artifacts

Per-run artifacts preserve the evidence for a single replication. They include the effective configuration used by the run, tool-side logs, parsed tool timelines, resolved IaC outputs, cloud observations, computed metrics, cleanup information, and status information. These artifacts make individual runs inspectable and support debugging when a run fails or when an observation is incomplete.

Per-run artifacts are especially important for lifecycle-aware benchmarking because a single aggregate duration is not enough to explain a run. The framework must preserve the intermediate evidence needed to determine whether a delay came from tool execution, cloud readiness, a specific managed service, or a cleanup issue.

Artifact level	Examples	Evaluation role
Raw run evidence	Tool logs, effective variables, resolved outputs, cloud observations, cleanup status	Makes a single replication inspectable and debuggable.
Derived run metrics	Apply duration, deployment time-to-ready, readiness lag, resource spans, observation-quality flags	Provides the run-level values used for aggregation.
Aggregated summaries	Per-configuration means, medians, spread statistics, success and failure counts	Supports Experiment 1 end-to-end comparison and variability analysis.
Graph-linked artifacts	Normalized graphs, runtime-annotated graphs, timeline charts, runtime-dominant dependency-path metadata	Supports Experiment 2 dependency-based explanation.

Table 4.5: Output artifact pipeline from raw run evidence to evaluation-ready summaries and graph-linked explanations.

4.9.2 Aggregated Summaries

Aggregated summaries combine repeated runs of the same experiment configuration. They record the number of successful and failed replications and aggregate timing values across runs. These summaries provide the bridge between individual run evidence and the evaluation-level comparisons in Chapter 6.

The summary pipeline supports repeated experiments by keeping all replications associated with the same manifest-driven configuration. This makes it possible to compare typical timing, variability, and failure patterns across tool-provider-scenario combinations.

4.9.3 Visualization and Graph Artifacts

The post-run analysis workflow produces timeline artifacts, normalized scenario-resource graph artifacts, runtime-annotated graph artifacts, runtime-dominant dependency-path metadata, and graph-linked analysis artifacts. Timeline artifacts show the ordering and overlap of resource-level provisioning observations. Normalized scenario-resource graph artifacts show the dependency structure produced after cleaning and normalization. Runtime-annotated graph artifacts connect these two views by attaching timing information to graph nodes. Runtime-dominant dependency-path metadata and graph-linked analysis artifacts support dependency-based reasoning.

These artifacts are used for dependency-based explanation rather than only presentation. A timeline can reveal overlap and delay, but not dependency structure. A graph can reveal dependency structure, but not timing. The combined artifacts allow the evaluation to interpret timing differences in terms of resource roles, dependency paths, and dependency chains.

4.9.4 Artifacts for Evaluation

The artifacts produced by the framework are consumed by the experimental setup and evaluation chapters. Chapter 5 defines the experimental factors, controlled conditions, metrics, and aggregation rules. Chapter 6 uses the framework outputs to compare deployment time-to-ready, readiness lag, variability, and dependency-based explanations.

The output pipeline is therefore part of the framework design, not merely an implementation convenience. Without structured outputs, repeated results could not be aggregated consistently, and dependency-based artifacts could not be connected to end-to-end measurements.

4.10 Reproducibility and Robustness

Reproducibility and robustness are central to the framework because cloud benchmarking is affected by nondeterminism, provider-side delays, transient failures, and cleanup challenges. The framework addresses these issues through manifest-driven execution, recorded run artifacts, repeated replications, explicit cleanup handling, and extensible component boundaries.

4.10.1 Manifest-Driven Reproducible Execution

Manifest-driven execution makes each experiment configuration explicit. The manifest defines the selected tool, provider, scenario, variables, replication count, timeout settings, and verification policy. During execution, the framework records the effective configuration and stores it with the run artifacts. This makes runs traceable and allows the same configuration to be rerun later.

The framework also uses copied templates and isolated run contexts to reduce interference between replications. This is important because repeated provisioning runs should differ because of normal execution variability and cloud-side nondeterminism, not because of manual configuration drift or uncontrolled template changes.

4.10.2 Failure Handling and Cleanup

The framework records failures as part of the benchmark evidence. If a run fails during initialization, apply execution, output resolution, cloud observation, or cleanup, the framework stores the error information and any available logs or partial observations. This supports inspection rather than hiding failed runs.

Cleanup is handled explicitly because cloud resources can remain in intermediate states or be temporarily blocked by provider-side dependencies. The framework therefore supports repeated destroy attempts, provider-specific backoff, post-destroy state inspection, and cleanup status reporting. If cleanup is incomplete, the run artifacts record this fact so that the evaluation can account for it.

Observation quality checks provide an additional robustness mechanism. They identify missing request timestamps, missing readiness timestamps, or unmatched tool references that would weaken later runtime annotation. This helps prevent dependency-based analysis from silently relying on incomplete observations.

4.10.3 Extensibility

The framework is designed to be extensible along the same dimensions used in the evaluation. New IaC tools can be supported by adding a tool driver that implements the common lifecycle responsibilities. New cloud providers can be supported by adding an observer that implements the common observation abstraction. New scenarios can be added through IaC templates and scenario specification files that encode the required resource roles, output mappings, cloud identities, and readiness predicates. New experiment combinations can be added through manifests.

This extensibility is important because the thesis evaluates a selected matrix of tools, providers, and scenarios, but the framework is intended to support future comparative studies. The framework therefore separates experiment definition, tool execution, cloud observation, scenario resolution, metric computation, and dependency analysis into independent responsibilities. This separation allows the framework to be extended without changing the overall benchmark lifecycle.

5

Experimental Setup

This chapter defines the experimental setup used to evaluate IaC provisioning behavior across tools, cloud providers, and benchmark scenarios. Chapter 3 defined the dependency-graph analysis method, and Chapter 4 described the lifecycle-aware benchmarking framework that implements this method. This chapter therefore does not introduce a new method or framework component. Instead, it operationalizes **RQ3** by defining the experimental factors, controlled conditions, metrics, data artifacts, quality checks, and analysis procedures used in Chapter 6.

The evaluation combines two complementary analyses. Section 5.6 defines the end-to-end provisioning analysis, which compares deployment time-to-ready, tool-side apply time, readiness lag, and run-to-run variability across all tool-provider-scenario configurations. Section 5.7 defines the dependency-based provisioning analysis, which connects selected end-to-end results to normalized dependency graphs, resource-level timeline artifacts, and runtime-dominant dependency paths. Together, these two analyses support both comparative benchmarking and structural explanation.

5.1 Evaluation Goals and Experiment Overview

This section defines how the evaluation is organized to answer **RQ3**. The evaluation separates repeated-run comparison from dependency-based explanation because the two parts require different evidence: aggregated summaries for timing and variability, and graph-plus-timeline artifacts for structural interpretation.

5.1.1 Evaluation Goals

The goal of this chapter is to define how the empirical evaluation answers **RQ3**. This question has two parts. The first part is comparative: it asks how Terraform, OpenTofu, and Pulumi behave across AWS, Azure, and GCP for the selected benchmark-inspired scenarios. This requires repeated measurements of deployment time-to-ready, apply time, readiness lag, and variability. The second part is explanatory: it asks how the observed differences can be interpreted through dependency graphs, resource-level timing, and runtime-dominant dependency paths. This requires connecting runtime observations to the normalized scenario-resource graphs defined in Chapter 3.

The evaluation therefore has three goals. First, it compares typical provisioning performance across the full benchmark matrix. Second, it compares variability across repeated runs, because a configuration that is occasionally fast but unstable is different from one that is consistently fast. Third, it uses dependency-based artifacts to explain selected timing patterns in terms of graph structure, resource-level timing, and runtime-dominant dependency paths.

5.1.2 Experiment Overview

The evaluation is divided into two experiments.

Experiment 1: End-to-end provisioning analysis. This experiment uses the aggregated summary outputs produced by the framework to compare provisioning behavior across all tool-provider-scenario combinations. It focuses on deployment time-to-ready, tool-side apply time, readiness lag, and variability across repeated runs. This experiment identifies the main performance and variability patterns in the result set.

Experiment 2: Dependency-based provisioning analysis. This experiment explains selected results from Experiment 1 by using the dependency-analysis artifacts produced by the framework. It combines normalized dependency graphs, resource-level timeline charts, runtime annotations, and runtime-dominant dependency paths. The purpose is not to replace the end-to-end comparison, but to explain why selected tool-provider-scenario combinations behave as they do.

The two experiments therefore play different roles. Experiment 1 identifies what happened at the benchmark-matrix level. Experiment 2 explains selected cases by connecting those outcomes to resource dependencies and resource-level provisioning timelines.

Factor	Option 1	Option 2	Option 3
IaC tool	Terraform	OpenTofu	Pulumi
Cloud provider	AWS	Azure	GCP
Resource scenario	YCSB-inspired	CloudSuite-inspired	DeathStarBench-inspired

Table 5.1: Experimental factors and options used in the benchmark matrix.

5.2 Experimental Factors and Benchmark Matrix

The evaluation varies three independent factors: IaC tool, cloud provider, and resource scenario. These factors correspond to the benchmark dimensions introduced in the thesis methodology and implemented by the framework manifests. Table 5.1 summarizes the evaluated factor values.

5.2.1 IaC Tool Factor

The IaC tool factor consists of Terraform, OpenTofu, and Pulumi. Terraform and OpenTofu are evaluated as Terraform-like declarative IaC tools with comparable command structures, while Pulumi is evaluated as an IaC tool with a different programming-language-based execution model. The purpose of the evaluation is not to compare language expressiveness or developer ergonomics. Instead, all three tools are treated as provisioning systems that instantiate semantically aligned infrastructure scenarios under the same benchmark lifecycle.

The framework interacts with these tools through the driver layer described in Chapter 4. Terraform and OpenTofu use the Terraform-like driver structure, while Pulumi uses a separate driver because its project and stack model differs from the Terraform-like tools. For fairness, the same high-level lifecycle is applied to all tools: initialization, apply execution, output resolution, cloud-side observation, metric computation, destroy, and cleanup.

The exact installed tool versions are treated as execution-environment metadata and are reported with the reproducibility material together with provider-plugin and package versions. The experiment design itself does not depend on a particular version number; however, reproducibility requires future reruns to document the versions used for Terraform, OpenTofu, Pulumi, Node.js, npm, and the relevant provider packages.

5.2.2 Cloud Provider Factor

The cloud provider factor consists of Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). These providers are evaluated as the target cloud environments in which the same benchmark-inspired scenarios are provisioned. The purpose of this factor is to measure how provisioning behavior differs across cloud providers when the IaC tool and resource scenario are held fixed.

The comparison uses the role-level scenario abstraction defined in Chapter 3. Therefore, the evaluation does not require concrete resources to be identical across providers. Instead, provider-specific implementations are treated as comparable when they satisfy the same scenario role within the benchmark scenario. The framework implementation described in Chapter 4 operationalizes this comparison through manifests, scenario specification files, provider-specific observers, and readiness checks.

Provider-specific differences in resource models, service behavior, API processing, and readiness semantics are not hidden by the abstraction. They are part of the measured outcome of the cloud-provider factor and are analyzed in Chapter 6.

5.2.3 Resource Scenario Factor

The resource scenario factor consists of three benchmark-inspired infrastructure scenarios: the YCSB-inspired data-serving scenario, the CloudSuite-inspired composite service scenario, and the DeathStarBench-inspired application-stack scenario. These are the same scenario instances defined methodologically in Chapter 3 and executed by the framework described in Chapter 4.

In this chapter, the scenarios are treated as experimental factor values rather than redescribed as new scenario designs. The YCSB-inspired scenario represents the compact data-serving case, the CloudSuite-inspired scenario represents the intermediate composite service case, and the DeathStarBench-inspired scenario represents the broadest application-stack case. Together, they allow the evaluation to compare provisioning behavior across increasing resource-role coverage, dependency complexity, and readiness observation complexity.

Table 5.2 summarizes the resource roles used by each scenario in the experimental matrix. The table is repeated here because these roles are the units used later to interpret resource-level timing, graph structure, and dependency-based results in Chapter 6.

Scenario	Resource roles
YCSB-inspired data-serving scenario	`network`, `subnet`, `vm`, `cache`
CloudSuite-inspired composite service scenario	`network`, `subnet`, `route`, `public_ingress`, `vm`, `network_interface`, `persistent_disk`
DeathStarBench-inspired application-stack scenario	`network`, `subnet`, `route`, `nat`, `public_ingress`, `cluster`, `node_pool`, `cache`, `database`, `container_registry`

Table 5.2: Resource roles used by the three benchmark-inspired scenarios.

Dimension	Count	Description
IaC tools	3	Terraform, OpenTofu, Pulumi
Cloud providers	3	AWS, Azure, GCP
Resource scenarios	3	YCSB-inspired, CloudSuite-inspired, DeathStarBench-inspired
Tool-provider-scenario configurations	27	Full factorial combination of the three factors
Successful replications per configuration	5	Repeated executions under the same manifest-defined configuration
Successful runs in main result set	135	Main dataset used for end-to-end timing and variability analysis

Table 5.3: Benchmark matrix and replication structure.

5.2.4 Full Factorial Matrix and Replications

The evaluation uses the full factorial combination of the three experimental factors:

$$3 \text{ tools} \times 3 \text{ providers} \times 3 \text{ scenarios} = 27 \text{ configurations}.$$

Each configuration is executed with five successful replications, yielding the main result set of 135 successful runs:

$$27 \text{ configurations} \times 5 \text{ successful replications} = 135 \text{ successful runs}.$$

The framework represents each configuration as a manifest under the experiment configuration directory. Each manifest specifies the scenario, provider, IaC tool, template path, scenario specification file, input variables, replication count, timeout settings, destroy-attempt limits, and verification settings. Repeated runs of the same manifest are treated as repeated measurements of the same tool-provider-scenario configuration.

Table 5.3 summarizes the matrix size and replication structure.

Controlled aspect	How it is controlled	Purpose
Benchmark lifecycle	All runs follow the same framework lifecycle: initialize, apply, resolve outputs, observe readiness, compute metrics, destroy, and store artifacts.	Ensures comparable measurement stages across tools and providers.
Experiment configuration	Each configuration is defined by a manifest containing the tool, provider, scenario, template path, variables, replications, timeouts, and verification settings.	Makes each run traceable and reproducible.
Scenario role mapping	Concrete provider resources are interpreted through shared scenario roles.	Supports fair cross-cloud comparison without claiming identical resource implementations.
Provider location	AWS uses `eu-west-1`; Azure uses `belgiumcentral`; GCP uses `europe-west1` and `europe-west1-b` where zonal resources are required.	Reduces uncontrolled regional variation within each provider.
Resource sizing	VM and node sizes are held consistent within each provider and scenario where applicable.	Reduces configuration drift between tools for the same provider-scenario cell.
Replication count	Each configuration contributes five successful replications to the main result set.	Enables variability analysis rather than single-run comparison.
Naming isolation	Runs use generated or manifest-derived names to avoid collisions between replications.	Reduces interference between repeated runs.
Cleanup policy	Destroy is executed after each replication; cleanup status is checked before the next accepted run.	Prevents resource accumulation and avoids including runs affected by unresolved cleanup or configuration errors.
Timeout policy	Timeouts and destroy-attempt limits are manifest-defined and held constant within each configuration.	Prevents ad hoc manual intervention during repeated runs.
Observation quality	Missing timestamps, unmatched resources, and incomplete graph-timeline alignment are flagged.	Prevents incomplete observations from being silently treated as complete evidence.

Table 5.4: Controlled conditions used in the experimental setup.

5.3 Controlled Conditions and Fairness Rules

The evaluation compares different tools and cloud providers, but it does not assume that their concrete implementations are identical. Fairness is therefore enforced through con-

trolled scenario roles, manifest-defined variables, common lifecycle stages, repeated executions, and explicit data inclusion rules. Table 5.4 summarizes the main controlled aspects.

5.3.1 Common Benchmark Lifecycle

Every run follows the standardized lifecycle implemented by the framework. The run begins with initialization and environment setup. The selected IaC tool is then initialized and used to execute the provisioning operation. After the apply operation, the framework resolves IaC outputs, maps them to tracked scenario resources, observes cloud-side readiness through provider-specific observers, computes metrics, destroys the provisioned resources, and stores structured artifacts.

This common lifecycle is important because Terraform, OpenTofu, and Pulumi expose different command structures and event formats, and AWS, Azure, and GCP expose different APIs and readiness states. The framework does not eliminate these differences. Instead, it measures each run through the same high-level stages so that differences in provisioning behavior can be compared under a consistent observation model.

5.3.2 Resource Configuration Alignment

Resource configuration alignment is enforced at the scenario-role level. For each scenario, the same role vocabulary is used across tools and cloud providers, while concrete provider resources are allowed to differ according to each provider's resource model. This follows the role-level scenario abstraction defined in Chapter 3 and implemented by the framework through the scenario specification files described in Chapter 4.

Within a provider-scenario configuration, the IaC tools use semantically aligned templates. The same major scenario roles are included where the provider model allows it, and VM or node sizes are kept comparable within each provider and scenario. For the YCSB-inspired data-serving scenario and the CloudSuite-inspired composite service scenario, AWS uses `t3.micro` virtual machines, Azure uses `Standard_D2s_v5`, and GCP uses `e2-standard-2`. For the DeathStarBench-inspired application-stack scenario, AWS uses `m7i-flex.large` EKS worker nodes, Azure uses `Standard_D2s_v5` AKS node pools, and GCP uses `e2-standard-2` GKE node pools. Managed service configurations are also kept compact to support repeated execution.

The goal of this alignment is not to make provider resources identical. Rather, the goal is to keep the scenario role and resource scale comparable while preserving provider-specific resource behavior as part of the empirical result. The detailed provider-specific

mapping from scenario roles to concrete AWS, Azure, and GCP resources is reported in Appendix B.1.

5.3.3 Execution Environment Controls

The benchmark framework is executed from a controlled runner environment with the required IaC tools, Python dependencies, Pulumi dependencies, cloud credentials, and optional graph-rendering tools installed. The framework implementation requires Python 3.10 or newer. Pulumi experiments additionally require Node.js and npm. Graph visualization requires Graphviz when SVG rendering is needed.

Cloud credentials are configured before the experiment begins. AWS runs use the AWS profile specified in the manifest, while Azure and GCP runs use credential files referenced by the manifests. The framework treats the selected cloud account, subscription, or project as part of the execution environment and records provider context where supported.

Timeouts are manifest-defined. Configurations for the YCSB-inspired data-serving scenario and most CloudSuite-inspired composite service scenario runs use shorter apply and destroy limits, while the DeathStarBench-inspired application-stack scenario uses longer limits because it includes slower managed-service components such as Kubernetes clusters, node pools, NAT, managed databases, and container registries. These limits are held constant within each tool-provider-scenario configuration. Therefore, timeout policy is controlled at the manifest level rather than manually adjusted per run.

5.3.4 Replication, Naming, and Cleanup Controls

Each tool-provider-scenario configuration is executed until five successful replications are available for the final result set. Repetition is necessary because cloud provisioning can vary due to provider-side scheduling, resource availability, API behavior, and readiness stabilization. A single run is therefore not treated as representative of the configuration.

To avoid collisions between repeated runs, the framework creates isolated run contexts and uses unique resource names or generated suffixes where required. This is particularly important for global or semi-global resources, managed service names, registry names, and cloud resources that may remain temporarily reserved after deletion.

Cleanup is performed after each replication through the selected IaC tool's destroy operation. This cleanup step is part of the experimental control because the benchmark measures provisioning from a clean scenario state rather than provisioning from existing scenario resources. The framework records destroy attempts, destroy duration, cleanup

status, and any error information so that incomplete cleanup can be detected before the next run. If a run fails because of a framework error, configuration error, credential issue, quota problem, or cleanup problem, the issue is investigated and corrected before the experiment is restarted. Such failed or debugging runs are not included in the final timing analysis. The final result set therefore consists only of configurations for which the required five successful replications were completed from a clean scenario state under the corrected experimental setup.

Metric or statistic	Definition	Purpose
`iac_apply_total_s`	Elapsed time of the IaC apply or update operation.	Measures tool-side provisioning duration.
`deployment_time_from_iac_apply_start_s`	Time from IaC apply start to the latest required cloud-observed readiness point.	Measures end-to-end deployment time-to-ready.
Readiness lag	Difference between deployment time-to-ready and tool-side apply duration.	Measures post-apply stabilization after tool-reported completion.
Tool-side resource span	Resource-level start, finish, and duration parsed from IaC tool events where available.	Supports resource-level timeline and apply-phase analysis.
Cloud provisioning latency	Time from provider-side request observation to cloud-observed readiness, where available.	Separates cloud-side provisioning behavior from tool-side execution.
Resource readiness lag	Time between tool-side resource completion and cloud-observed resource readiness, where available.	Identifies resources that require post-tool stabilization.
Mean and median	Central tendency across replications.	Describes typical behavior.
Standard deviation and IQR	Absolute spread across replications.	Describes run-to-run variability in seconds.
Coefficient of variation	Standard deviation divided by the mean.	Compares relative variability across configurations with different runtimes.
Success and failure count	Number of successful and failed replications in a configuration.	Describes execution outcome and data-inclusion context.

Table 5.5: Operational metrics used in the evaluation.

5.4 Measurement Model and Metrics

The evaluation uses the measurement model implemented by the framework in Chapter 4. The framework records raw observations from the IaC tool and from cloud-provider observers, then derives run-level, resource-level, and aggregate metrics from these observations. Table 5.5 summarizes the main metrics used in the evaluation.

5.4.1 Raw Observations

Raw observations are the timestamps and status records collected before metric aggregation. On the tool side, the framework records the apply start time, apply completion time, and resource-level start and finish events parsed from the IaC tool event stream. On the cloud side, the framework records provider-side request timestamps, resource creation timestamps, and readiness timestamps from scenario-specific readiness checks.

The availability and precision of these observations can differ across tools, providers, and resource types. The framework therefore performs observation-quality checks during run execution. In the current implementation, these checks identify missing cloud request timestamps, missing cloud creation timestamps, missing cloud readiness timestamps, and scenario resources whose tool reference cannot be matched to the IaC timeline. Critical gaps, such as missing request timestamps, missing readiness timestamps, or unmatched tool references, cause the run to fail rather than being silently included in the final result set. Such issues are corrected before the experiment is rerun, so the final analysis uses accepted runs with the required observations available.

5.4.2 End-to-End Metrics

The primary end-to-end metric is deployment time-to-ready, stored as:

`deployment_time_from_iac_apply_start_s`.

This metric measures the elapsed time from the start of the IaC apply operation to the point at which all tracked readiness conditions for the scenario have been satisfied. It is the primary performance metric because the thesis studies practical infrastructure usability rather than only tool command completion.

The main tool-side metric is:

`iac_apply_total_s`.

This metric measures the duration of the IaC tool's apply or update operation. It captures the tool-observed provisioning interval but does not necessarily imply that every tracked resource is ready for use.

The end-to-end readiness lag is derived as:

$$readiness\_lag = \texttt{deployment\_time\_from\_iac\_apply\_start\_s} - \texttt{iac\_apply\_total\_s}.$$

A positive readiness lag indicates that the framework observed additional stabilization time after the IaC tool reported apply completion. This value is important because it captures the part of the provisioning lifecycle that would be missed by a benchmark that stops at tool-reported completion.

5.4.3 Lifecycle Decomposition Metrics

Lifecycle decomposition separates tool-side execution from cloud-side stabilization. The apply duration captures the IaC tool's execution phase. The readiness lag captures the post-apply stabilization phase. Where provider-side request and creation timestamps are available, the evaluation also considers cloud-side phase durations.

For a resource r, cloud-side provisioning latency can be expressed as:

$$cloud\_ready\_from\_request(r) = t_{cloud\_ready}(r) - t_{cloud\_request}(r).$$

Similarly, resource-level readiness lag can be expressed as:

$$cloud\_ready\_from\_iac\_end(r) = t_{cloud\_ready}(r) - t_{iac\_end}(r).$$

These resource-level decomposition metrics are used when the necessary timestamps are available and when the resource can be matched to both a tool-side reference and a cloud-side observation. They support analysis of whether delays are dominated by tool execution, provider-side provisioning, or post-apply readiness stabilization.

5.4.4 Resource-Level Timing Metrics

Resource-level timing metrics are used in two ways. First, the all-resource timeline and runtime-attached dependency graphs use tool-side resource spans. For each resource that appears in the parsed IaC apply timeline, the framework records a start offset, finish offset, and duration relative to the apply start time. These values are used to align timeline rows with graph nodes and to compute runtime-dominant dependency paths. This follows

the implementation boundary described in Section 4.8.3: tool-side resource spans align directly with the IaC-derived graph nodes, whereas cloud-side readiness observations are not available at the same fine-grained graph-node granularity.

Second, managed-resource timing summaries preserve cloud-side request, creation, and readiness timestamps where they are available. These cloud-side observations support lifecycle decomposition, readiness-lag analysis, and interpretation of post-apply stabilization, but they are not the finish metric used by the current runtime-dominant dependency-path algorithm.

The tool-side resource finish offset is especially important for runtime-dominant dependency-path identification. As defined in Chapter 3 and implemented in Chapter 4, the current implementation starts from the graph node with the latest IaC apply finish offset and walks backward through predecessors using the latest predecessor finish offset at each step.

5.4.5 Aggregation and Variability Statistics

For each tool-provider-scenario configuration, the evaluation aggregates the five successful replications. Mean and median are used to describe typical performance. Standard deviation and interquartile range are used to describe absolute run-to-run variability. The coefficient of variation is used to compare relative variability across configurations with different runtime scales.

The coefficient of variation is defined as:

$$CV = \frac{\sigma}{\mu},$$

where σ is the standard deviation and μ is the mean of the measured values. This statistic is useful because a 20-second standard deviation has different meaning for a 100-second deployment than for an 850-second deployment. The evaluation therefore reports both absolute variability and relative variability.

5.5 Data Collection, Quality Checks, and Aggregation

The framework produces artifacts at three levels: per-run artifacts, aggregated summaries, and post-run analysis artifacts. Chapter 4 describes how these artifacts are produced. This section defines how they are used in the evaluation.

Table 5.6 summarizes the role of the main artifact categories before the following subsections describe how they are collected, checked, and included. The table also clarifies

Artifact type	Experiment 1	Experiment 2
Aggregated summary JSON files	Main input for timing and variability analysis	Used for resource-level summary support
Per-run metric values	Used to compute distributions and variability	Used when selecting representative runs
Success and failure counts	Used for inclusion and execution-outcome context	Used to qualify selected cases
Resource-level variability summaries	Used for resource-level timing spread	Used to support graph and timeline interpretation
All-resource timeline charts	Not a primary input	Main visual artifact for temporal ordering and overlap
Dependency graph SVG artifacts	Not a primary input	Main visual artifact for graph-structure interpretation

Table 5.6: Artifact types used by the two evaluation experiments.

why the two experiments use different evidence: Experiment 1 depends primarily on aggregated repeated-run summaries, while Experiment 2 depends on graph and timeline artifacts supported by those summaries.

The artifact-use distinction prevents the evaluation from treating a single visual artifact as statistical evidence. Repeated-run summaries provide the basis for timing and variability claims, while selected graphs and timelines explain representative cases in relation to dependency structure.

5.5.1 Per-Run Artifacts

A per-run artifact set preserves the evidence for one replication. It includes the effective configuration, tool-side logs, parsed tool timeline, resolved IaC outputs, cloud observations, computed metrics, cleanup status, and observation-quality information. These artifacts make individual runs inspectable and allow the evaluation to trace aggregate values back to the run-level evidence.

For Experiment 1, per-run metric values are used to compute central tendency and variability across replications. For Experiment 2, per-run timeline and graph-linked artifacts are used to inspect resource ordering, overlap, and runtime-dominant dependency paths. A single representative run is not treated as evidence of variability by itself; it is interpreted together with repeated-run summaries from the aggregated result files.

5.5.2 Aggregated Summary Files

The main repeated-run input for Experiment 1 is the summary bundle. It contains one aggregated summary JSON file for each tool-provider-scenario configuration, for a total of 27 summary files. Each file groups five successful replications of the same manifest-defined configuration. The summary files store run-level values such as deployment time-to-ready and apply time, success and failure counts, and resource-level variability summaries where available.

These summary files are the main input for end-to-end comparison because they preserve the repeated-run values needed to compute mean, median, standard deviation, interquartile range, coefficient of variation, and min/max ranges. They also provide resource-level variability groups, including IaC apply duration by resource, cloud provisioning latency, readiness lag, and cloud phase durations where available.

5.5.3 Observation Quality Checks

Observation quality checks are used to prevent incomplete timing evidence from entering the final accepted result set. The framework checks whether required cloud request and readiness timestamps are present and whether scenario resources can be matched to tool-side timeline entries. If critical observation gaps are detected, the run is marked as failed. The underlying issue is then investigated and corrected before the experiment is rerun.

Non-critical gaps, such as missing creation timestamps for resources where request and readiness timestamps are still available, may limit some fine-grained decomposition metrics. However, critical gaps are not treated as valid timing evidence for the final analysis.

5.5.4 Data Inclusion Rules

The main end-to-end timing analysis includes only runs from the final accepted experiment set. A run is included when the apply operation completes successfully, the framework observes the required readiness condition for the scenario, and cleanup status does not indicate unresolved interference with later replications. Runs produced during debugging, configuration correction, or recovery from failed cleanup are excluded from the final timing distributions.

The dependency-based analysis uses the same final accepted experiment set as the end-to-end timing analysis, together with the graph and timeline artifacts generated from that set. A run is not treated as valid dependency-analysis evidence unless the required scenario resources can be matched to tool-side timeline entries and the corresponding graph and

timeline artifacts can be generated. If graph-timeline alignment or artifact generation fails during experimentation, the underlying issue is investigated and corrected before the experiment or artifact-generation workflow is rerun. Such incomplete or debugging outputs are not included in the final dependency-based analysis.

The evaluation also distinguishes between repeated-run summaries and representative visual artifacts. Summary JSON files provide the repeated-run statistical evidence. Timeline charts and graph SVGs provide run-level visual explanations. Therefore, visual interpretations in Chapter 6 are supported by repeated-run summaries rather than treated as standalone proof of variability.

5.6 Experiment 1: End-to-End Provisioning Analysis

Experiment 1 compares how Terraform, OpenTofu, and Pulumi provision the three benchmark-inspired scenarios across AWS, Azure, and GCP. The comparison focuses on deployment time-to-ready, tool-side apply time, readiness lag, and run-to-run variability. The purpose is to identify which factor combinations are faster, slower, more stable, or more variable under repeated execution.

5.6.1 Inputs

The inputs for Experiment 1 are the aggregated summary JSON files produced by the framework. These files contain the per-run metric values for each tool-provider-scenario configuration, manifest metadata, success and failure counts, and resource-level variability summaries. The main run-level fields used in this experiment are `deployment_time_from_iac_apply_start_s` and `iac_apply_total_s`. The readiness lag is derived from these values.

When available, resource-level variability summaries are used to support later interpretation of which resources contribute to observed timing patterns. However, the primary unit of comparison in Experiment 1 is the tool-provider-scenario configuration.

5.6.2 Comparison Dimensions

Experiment 1 compares the results across five analytical dimensions:

1. **Scenario-level comparison.** This comparison asks how provisioning behavior changes from the YCSB-inspired data-serving scenario to the CloudSuite-inspired

composite service scenario and the DeathStarBench-inspired application-stack scenario. It captures how scenario composition and dependency complexity affect deployment time-to-ready and variability.

2. **Provider-level comparison.** This comparison asks how AWS, Azure, and GCP differ for semantically aligned scenarios. It captures provider-specific service behavior, readiness semantics, and managed-service provisioning delays.

3. **Tool-level comparison.** This comparison asks how Terraform, OpenTofu, and Pulumi differ when provisioning the same provider-scenario combination. It captures tool-side execution differences under semantically aligned templates.

4. **Configuration-level comparison.** This comparison examines each of the 27 tool-provider-scenario configurations. It is the most precise comparison unit because it fixes all three experimental factors.

5. **Replication-level variability comparison.** This comparison examines how much repeated runs vary within the same configuration. It distinguishes configurations that are consistently fast from configurations that have unstable timing behavior.

5.6.3 Analysis Procedure

The analysis procedure for Experiment 1 consists of five steps:

1. **Compute central tendency and variability per configuration.** For each of the 27 tool-provider-scenario configurations, the evaluation computes mean, median, standard deviation, interquartile range, coefficient of variation, minimum, and maximum for deployment time-to-ready and apply time.

2. **Compare deployment time-to-ready across the full matrix.** This step identifies the fastest and slowest configurations and the main trends across scenarios, providers, and tools.

3. **Decompose deployment time-to-ready into apply time and readiness lag.** This step determines whether observed end-to-end time is dominated by tool-side execution or by post-apply cloud-side stabilization.

4. **Compare variability across repeated runs.** This step considers both absolute variability, using standard deviation and interquartile range, and relative variability, using the coefficient of variation.

5. **Identify informative configurations for dependency-based explanation.** This step selects configurations for Experiment 2, including the fastest configurations, slowest configurations, configurations with large readiness lag, configurations with high variability, and configurations that differ from the broader trend.

5.6.4 Evaluation Outputs

The outputs reported in Chapter 6 include a time-to-ready comparison across all 27 configurations, factor-level apply-time and readiness-lag decomposition, configuration-level variability analysis, and resource-level readiness summaries. The reported figures include a grouped bar chart of scenario-provider-tool mean time-to-ready values, a stacked factor-level apply/readiness-lag chart with coefficient-of-variation annotations, a readiness-lag-share decomposition for selected configurations, reduced main-text views of the highest- and lowest-variability configurations, and reduced resource-level charts for cloud request-to-ready duration and readiness lag. The complete ranking figures are provided in Appendix B.2.

5.7 Experiment 2: Dependency-Based Provisioning Analysis

Experiment 2 explains selected end-to-end results using dependency graphs, IaC-tool-side all-resource timeline charts, runtime annotations, and runtime-dominant dependency paths. Cloud-side readiness and readiness-lag summaries are used as supporting lifecycle evidence when interpreting whether the graph-level timing explanation is sufficient or whether additional post-apply stabilization must be considered.

This experiment follows the dependency-based reasoning method from Chapter 3. Structural analysis provides graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates. Runtime annotation attaches timing observations to graph nodes. Dependency-based reasoning then compares structural candidates with runtime-dominant dependency paths and timeline evidence.

5.7.1 Inputs

The inputs for Experiment 2 are the dependency-analysis artifacts generated by the framework workflow. In the result set used in this thesis, these artifacts are represented by exported dependency-graph SVG files and all-resource timeline charts for each of the 27 configurations. The graph artifacts provide the structural view of the provisioning scenario, while the timeline charts provide the temporal view of resource-level provisioning behavior.

This follows the implementation boundary described in Section 4.8.3. The dependency-based graph and timeline artifacts use tool-side resource spans because these timings align directly with the IaC-derived graph nodes. Cloud-side readiness observations are used as lifecycle evidence, but they are not available at the same fine-grained graph-node granularity and are therefore not used as the finish metric for runtime-dominant dependency-path identification.

Experiment 2 also uses resource-level timing summaries from the aggregated summary JSON files. These summaries provide repeated-run support for the visual interpretation, so that graph and timeline figures are not interpreted as standalone evidence of variability. The underlying machine-readable analysis artifacts are treated as part of the framework workflow that generates and verifies the exported graph and timeline outputs.

5.7.2 Case Selection Strategy

Because the full benchmark matrix contains 27 configurations, Experiment 2 focuses on selected cases rather than explaining every configuration in the same level of detail. The selected cases are chosen based on the results of Experiment 1.

The main selection criteria are as follows. First, slow configurations are selected when they represent important end-to-end performance bottlenecks. Second, fast configurations are selected when they provide useful contrast with slower configurations in the same scenario. Third, configurations with large readiness lag are selected because they reveal cases where tool-reported completion differs from practical readiness. Fourth, high-variability configurations are selected because they show where repeated runs are less predictable. Finally, at least one representative case per scenario may be included when space is limited, so that the dependency-based analysis covers the progression from compact data-serving infrastructure to application-stack infrastructure.

Selected timeline and graph artifacts are interpreted as representative run-level visualizations. Repeated-run summary statistics are used to support claims about typical behavior and variability.

5.7.3 Analysis Procedure

The dependency-based analysis follows six steps:

1. **Inspect the normalized graph structure.** This step identifies the scenario resources, dependency edges, and role-level structure of the selected configuration.

2. **Interpret structural properties.** This step examines graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates. It identifies which dependency chains may matter before runtime timing is considered.

3. **Align resource timing observations with graph nodes.** This step connects the all-resource timeline with the dependency graph through the identity model implemented by the framework.

4. **Identify runtime-dominant dependency paths.** Runtime-dominant dependency paths are identified for the selected representative runs using IaC tool-side finish offsets. The runtime-dominant dependency path is run-specific because it depends on observed tool-side finish offsets rather than graph depth alone.

5. **Compare structural candidates with runtime-dominant dependency paths.** If a structurally deep path also dominates the runtime timeline, then graph structure provides a strong explanation for the observed end-to-end timing. If a shorter path dominates runtime because it contains a resource with a slow tool-side resource span, then the explanation is more resource-specific than depth-specific.

6. **Interpret the source of observed delays.** The final step interprets whether the observed delays are mainly explained by graph structure, tool-side resource scheduling, slow tool-side resource spans for specific resources, or post-apply readiness delay observed in the lifecycle metrics. This links the graph and timeline evidence back to the end-to-end results from Experiment 1.

5.7.4 Evaluation Outputs

The outputs reported in Chapter 6 include selected graph-and-timeline pairs, repeated-run support for the selected cases, and a compact cross-case interpretation table. The evaluation relates the highlighted dependency paths in the dependency-graph artifacts to the all-resource timeline charts and to the repeated-run summaries. The graph and timeline figures are interpreted as representative first-replication explanations, while claims about typical behavior and variability remain grounded in the five-replication summary statistics.

6

Evaluation

This chapter evaluates IaC provisioning behavior using the experimental setup defined in Chapter 5. The evaluation answers RQ3 by combining repeated-run benchmark results with dependency-based explanation. Section 6.2 reports deployment time-to-ready, readiness behavior, and run-to-run variability across the full benchmark matrix. Section 6.3 then uses selected dependency graphs and all-resource timelines to explain how resource structure and tool-side resource spans contribute to representative outcomes. The chapter closes by synthesizing these findings into a direct answer to RQ3 and by discussing threats to validity.

6.1 Overview of the Result Set

The evaluation uses the full factorial benchmark matrix defined in Chapter 5: three resource scenarios, three cloud providers, and three IaC tools. This gives 27 tool-provider-scenario configurations. Each configuration contributes five successful replications to the final accepted result set, giving 135 successful provisioning runs for the repeated-run analysis.

6.1.1 Main Findings

The result set supports four main findings that connect the two experiments in this chapter.

1. **Scenario and provider effects dominate universal tool effects.** Experiment 1 shows that scenario complexity and provider-specific behavior explain larger time-to-ready differences than a global Terraform, OpenTofu, or Pulumi ranking. This finding is analyzed in Section 6.2.1 using Figures 6.1 and 6.2.

Artifact type	Count	Unit	Experiment 1	Experiment 2
Aggregated summary JSON files	27	One per tool-provider-scenario configuration	Primary input	Supporting evidence
Successful run records	135	Five replications per configuration	Primary input	Supporting evidence
All-resource timeline charts	27	First successful replication per configuration	Not primary input	Primary explanatory artifact
Runtime-attached dependency graph SVG files	27	First successful replication per configuration	Not primary input	Primary explanatory artifact

Table 6.1: Result-set and artifact inventory used in Chapter 6.

2. **Apply time usually dominates, but readiness lag still matters.** Experiment 1 shows that most deployment time-to-ready is spent inside the IaC apply phase, but selected configurations still have a visible post-apply readiness component. This finding is analyzed in Section 6.2.2 using Figures 6.2 and 6.3.

3. **Fast configurations are not always the most stable.** Experiment 1 shows that relative run-to-run variability is not determined by mean deployment time alone. Some slow configurations are stable, while some fast configurations are relatively variable. This finding is analyzed in Section 6.2.3 using Figure 6.4.

4. **Dependency graphs and timelines explain why the aggregate results occur.** Experiment 2 shows that graph depth alone does not determine deployment time-to-ready. A compact graph can be slow when one managed resource dominates timing, a fast mid-sized graph can still be readiness-lag sensitive, and a large application-stack graph can accumulate several long managed-service windows. This finding is analyzed in Section 6.3.3 and the case studies in Sections 6.3.4–6.3.6.

6.1.2 Artifact Scope

The result artifacts are used differently by the two experiments. Experiment 1 uses the aggregated summary files and successful run records as the primary evidence for time-to-ready, apply time, readiness lag, and run-to-run variability. Experiment 2 uses dependency-graph SVG files and all-resource timeline charts as explanatory artifacts for selected runs. The important boundary is that repeated-run summary statistics support claims about typical behavior and variability, while graph and timeline artifacts support resource-level explanation.

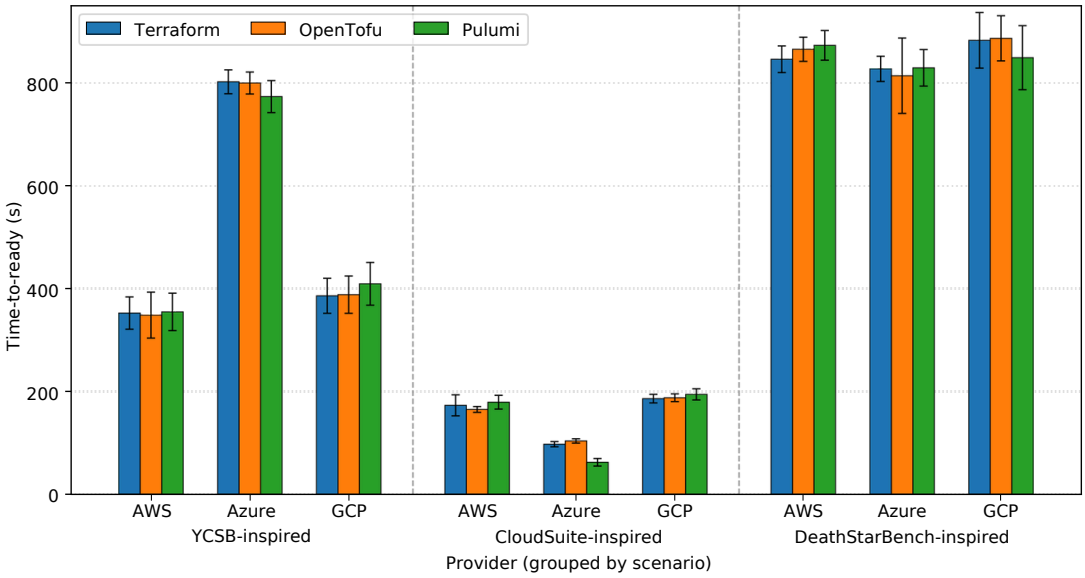

Figure 6.1: Mean deployment time-to-ready across the full benchmark matrix. Bars are grouped by benchmark-inspired scenario and cloud provider, with one bar per IaC tool. Error bars show run-to-run variability across the five successful replications for each tool-provider-scenario configuration.

6.2 Results for Experiment 1: End-to-End Provisioning Analysis

Experiment 1 provides the quantitative comparison across the full benchmark matrix. The subsections are ordered from the most important aggregate findings to the finer resource-level interpretation: factor effects first, apply-versus-readiness decomposition second, variability third, and resource-level readiness patterns fourth.

6.2.1 Scenario and Provider Effects Dominate Tool Effects

The most important end-to-end finding is that scenario and provider effects dominate universal IaC-tool effects. Figure 6.1 shows the mean deployment time-to-ready across all 27 tool-provider-scenario configurations. The horizontal axis groups the results first by benchmark-inspired scenario and then by cloud provider. The vertical axis reports deployment time-to-ready in seconds. Within each provider group, the bars compare Terraform, OpenTofu, and Pulumi, and the error bars show variation across the five successful replications.

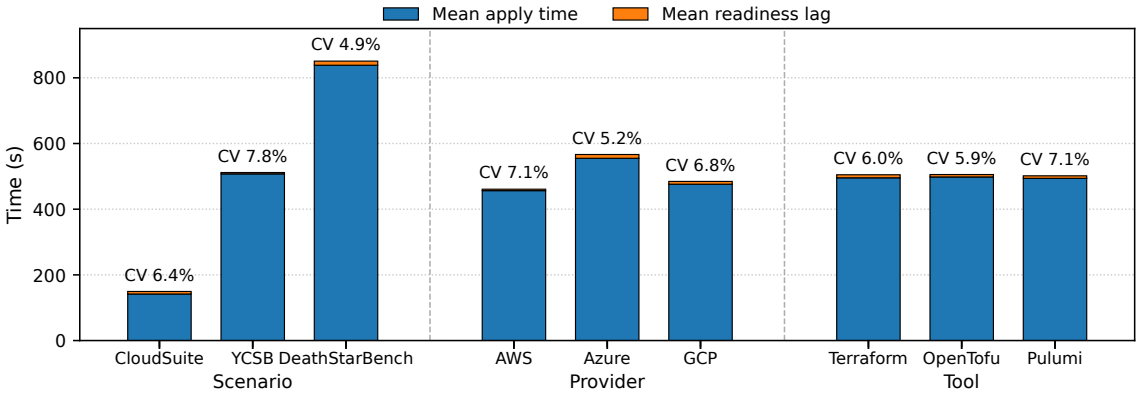

Figure 6.2: Factor-level averages for deployment time-to-ready. Each stacked bar decomposes the mean time-to-ready into mean IaC apply time and mean readiness lag for one scenario, provider, or tool factor level. The annotation above each bar reports the mean coefficient of variation (CV), showing relative run-to-run variability for that factor level.

The numbers show a large spread across the matrix. The fastest configuration is the CloudSuite-inspired scenario on Azure with Pulumi, with a mean time-to-ready of 62.1 s. The slowest configuration is the DeathStarBench-inspired scenario on GCP with OpenTofu, with a mean time-to-ready of 884.9 s. This means that the benchmark matrix does not support a single global ranking of tools or providers. Instead, the effect of a provider or tool depends on the scenario being provisioned.

Figure 6.2 aggregates the same result set by scenario, provider, and tool. Overall, the figure shows factor-level mean time-to-ready and its decomposition into apply time and readiness lag. The horizontal axis lists factor values: the three scenarios, the three providers, and the three IaC tools. The vertical axis shows mean time-to-ready in seconds. Each stacked bar shows the mean apply time in blue and the mean readiness lag in orange across the relevant five-run configuration summaries, and the label above each bar reports the mean coefficient of variation.

At the scenario level, the CloudSuite-inspired scenario averages 149.5 s, the YCSB-inspired scenario averages 511.6 s, and the DeathStarBench-inspired scenario averages 850.9 s. At the provider and tool levels, the aggregated bars are less clearly ordered. Changing the provider within the same scenario-tool pair changes time-to-ready by an average range of 199.2 s, whereas changing the IaC tool within the same scenario-provider pair changes it by an average range of 22.4 s. This means that scenario composition and provider-specific service behavior explain most of the observed differences. Tool choice still

matters in individual cells, such as CloudSuite Azure Pulumi and DeathStarBench GCP Pulumi, but the results do not show one IaC tool that dominates across the whole matrix.

6.2.2 Apply Time Usually Dominates Readiness Lag

The second finding is that apply time usually dominates deployment time-to-ready, although readiness lag remains important in selected cases. Figure 6.2 shows this at the factor level: most stacked bars are mostly blue, meaning that most measured deployment time is spent inside the IaC apply operation rather than after the apply operation has returned. Numerically, the factor-level averages also show this pattern. For example, the CloudSuite-inspired scenario averages 149.5 s time-to-ready, of which 141.7 s is apply time and 7.8 s is readiness lag; the DeathStarBench-inspired scenario averages 850.9 s, of which 838.3 s is apply time and 12.5 s is readiness lag.

This means that, for most configurations, the main contributor to end-to-end provisioning time is not a long deployment-level post-apply wait. Instead, the tool-side apply phase already includes much of the provider-side waiting required before the IaC tool reports completion. This supports the lifecycle distinction introduced earlier in the thesis: apply completion and cloud-observed readiness should be measured separately, but their relative sizes differ by configuration.

To keep the main-text figures readable, Figures 6.4–6.6 show reduced views of the full ranking outputs. Figure 6.4 keeps the most and least variable configurations, while Figures 6.5 and 6.6 keep the ten largest resource-level entries. The complete rankings are provided in Appendix B.2.

Figure 6.3 focuses on the configurations with the largest readiness-lag share. Overall, it shows the cases where post-apply stabilization is most visible in the end-to-end metric. The horizontal axis shows deployment time-to-ready in seconds, and the vertical axis lists the selected configurations. Each stacked bar shows mean apply time in blue and mean readiness lag in orange across five successful replications.

The strongest example is CloudSuite Azure Pulumi. It is the fastest configuration overall, with a mean time-to-ready of 62.1 s, but readiness lag still accounts for 16.5% of that total. CloudSuite AWS Terraform is also notable because its readiness-lag component is small in absolute terms but has high relative variability. These numbers mean that readiness lag is not the dominant component in most deployments, but it can still affect interpretation, especially for short-running configurations where a small absolute delay becomes a visible share of total time. A likely reason is that some resources finish tool-side provisioning

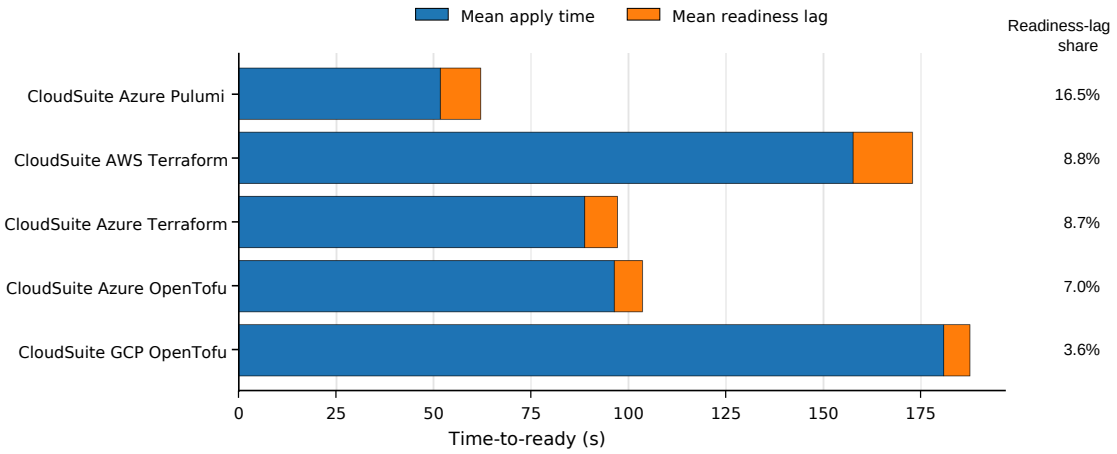

Figure 6.3: Configurations with the largest readiness-lag share. Each stacked bar decomposes mean deployment time-to-ready into mean IaC apply time and mean post-apply readiness lag. The figure highlights cases where readiness lag is small in absolute terms but still accounts for a visible share of total time-to-ready.

quickly but continue through provider-side stabilization before they satisfy the framework's readiness predicate.

6.2.3 Fast Configurations Can Still Be Variable

The third finding is that mean time-to-ready and run-to-run stability are different properties. Figure 6.4 shows the five most variable and five least variable configurations by coefficient of variation of deployment time-to-ready. Overall, the figure emphasizes relative variability rather than absolute duration. The horizontal axis shows the coefficient of variation in percent, while the vertical axis lists selected tool-provider-scenario configurations from the highest-CV group and the lowest-CV group. The complete ranking across all 27 configurations is provided in Appendix B.2.

The largest absolute standard deviations appear mostly in the DeathStarBench-inspired scenario, where the total deployment time is long and managed services contribute large timing windows. The largest relative variability, however, appears in smaller or faster configurations, especially YCSB AWS OpenTofu, CloudSuite Azure Pulumi, and CloudSuite AWS Terraform. For example, YCSB Azure OpenTofu is one of the slowest YCSB configurations, but it has the lowest relative variability in the result set. Conversely, CloudSuite Azure Pulumi is the fastest configuration overall, but it is among the most variable in relative terms.

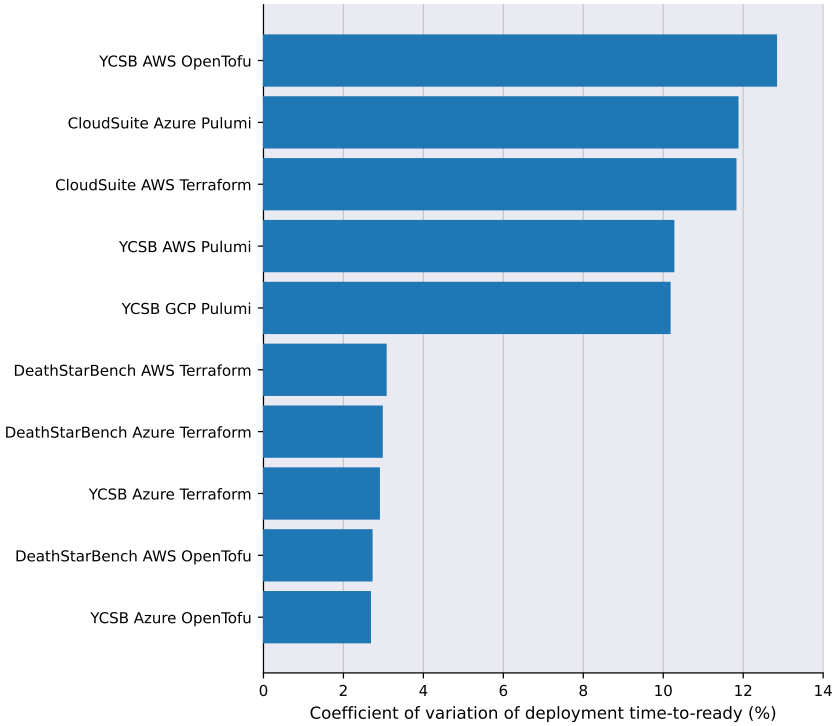

Figure 6.4: Most and least variable configurations by coefficient of variation of deployment time-to-ready. The figure shows the five highest-CV and five lowest-CV tool-provider-scenario configurations, computed from repeated successful runs. Middle-ranking configurations are omitted from the main text for readability; the full ranking across all configurations is provided in Appendix B.2.

This means that users should not interpret a low mean time-to-ready as evidence of predictability. A configuration can be fast on average but still less predictable across repeated runs. The likely explanation is that relative variability is sensitive to the size of the denominator: in short deployments, even a few seconds of scheduling, API, or readiness variation can produce a high coefficient of variation, while longer deployments can absorb similar absolute variation into a lower relative CV.

6.2.4 Resource-Level Readiness is Role- and Provider-Specific

The fourth finding is that aggregate deployment results are explained by different resource roles in different scenarios and providers. Figures 6.5 and 6.6 provide two complementary resource-level views. Figure 6.5 shows cloud request-to-ready duration, while Figure 6.6 shows resource-level readiness lag after tool-side completion. In both figures, the horizontal axis shows time in seconds, and the vertical axis lists scenario-provider-resource-role

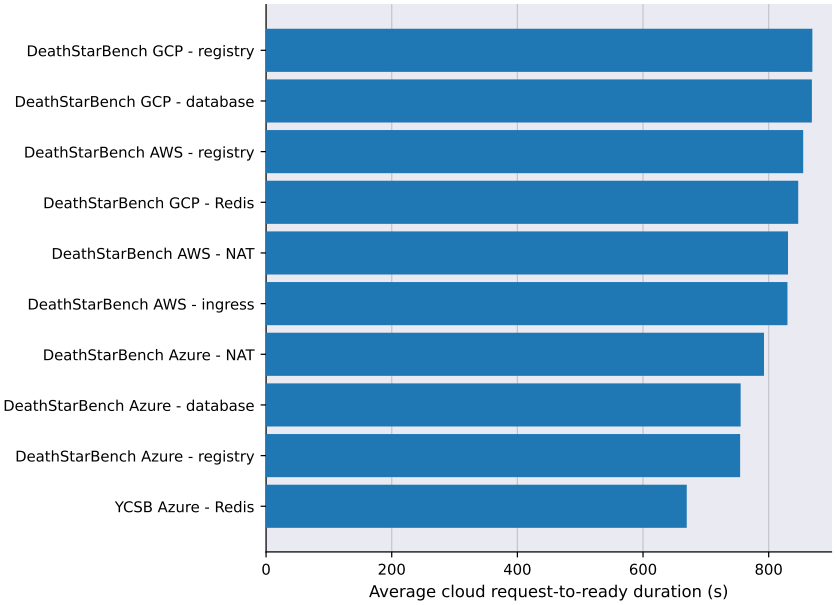

Figure 6.5: Largest resource-level cloud request-to-ready durations by scenario and provider, averaged across IaC tools. The figure shows the ten largest resource-role entries from the full resource-level ranking. It highlights which resource roles dominate cloud-side provisioning and readiness windows, while the complete ranking is provided in Appendix B.2.

combinations. Each bar is averaged across IaC tools.

The figures show that no single resource role explains all results. In the YCSB-inspired scenario, VM and Redis roles provide the largest resource-level timing signals. In the CloudSuite-inspired scenario, disk, VM, and public-ingress roles are most visible. In the DeathStarBench-inspired scenario, several managed-service and orchestration roles become important, including registry, NAT, ingress, database, Redis, and Kubernetes-related roles. For example, the DeathStarBench GCP registry appears near the top of both resource-level charts, while the YCSB Azure VM and Redis roles explain why the compact YCSB case can still be slow on Azure.

These numbers mean that end-to-end differences should not be attributed only to scenario size. They also reflect which provider-specific services implement the scenario roles and how those services stabilize. This is why Experiment 2 uses graph and timeline artifacts: the resource-level charts identify candidate resource roles, while dependency-based analysis shows how those roles sit inside the provisioning structure.

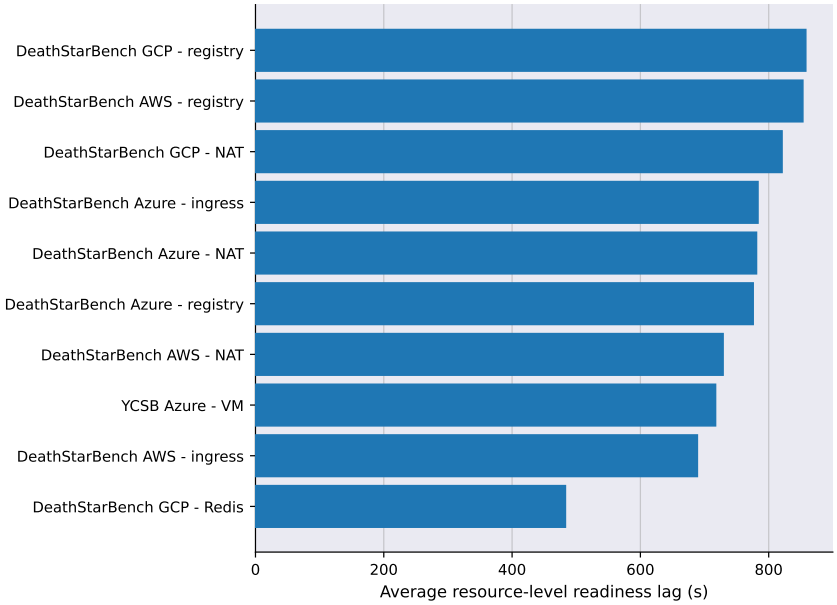

Figure 6.6: Largest resource-level readiness-lag durations by scenario and provider, averaged across IaC tools. The figure shows the ten largest resource-role entries whose cloud-side readiness occurred after tool-side completion. These values should be interpreted as resource-level lifecycle signals rather than directly additive components of deployment-level readiness lag; the complete ranking is provided in Appendix B.2.

6.2.5 Summary of Experiment 1 Findings

Experiment 1 shows four main findings. First, scenario and provider effects dominate universal IaC-tool effects. Second, apply time usually dominates deployment time-to-ready, although readiness lag remains important in selected configurations. Third, run-to-run variability must be interpreted separately from mean time-to-ready because fast configurations can still be relatively variable. Fourth, resource-level readiness behavior is role- and provider-specific. These observations motivate Experiment 2: the aggregate results identify what differs, while dependency graphs and timelines are needed to explain why those differences occur.

6.3 Results for Experiment 2: Dependency-Based Provisioning Analysis

Experiment 2 provides the dependency-based explanation of selected outcomes from Experiment 1. It does not repeat the full matrix comparison. Instead, it uses selected graph and timeline artifacts to explain how resource structure, tool-side resource spans, and cloud-

Case	Configuration	Selection reason
YCSB Azure case	YCSB-inspired, Azure, Terraform	Azure is much slower than AWS and GCP for the YCSB-inspired scenario
CloudSuite fast/variable case	CloudSuite-inspired, Azure, Pulumi	Fastest configuration overall, but with high relative variability and non-negligible lag share
DeathStarBench large-graph case	DeathStarBench-inspired, GCP, OpenTofu	Slowest configuration overall and a representative large dependency graph

Table 6.2: Selected cases for Experiment 2.

side readiness evidence relate to the repeated-run results. The subsections first define the case-selection boundary and reading guide, then present the main cross-case finding before discussing the individual cases.

6.3.1 Case Selection and Evidence Boundary

Experiment 2 selects three cases from the full result set, as summarized in Table 6.2. The cases cover the three scenario levels and represent three different observations from Experiment 1: a slow provider-specific YCSB result, a fast but relatively variable CloudSuite result, and the largest DeathStarBench result. The selected graph and timeline artifacts are used for structural and temporal explanation, while repeated-run statistics remain the basis for claims about typical behavior and variability.

6.3.2 How to Read the Graph and Timeline Artifacts

The graph-and-timeline figures should be read as paired explanations of one representative run. The graph at the top shows the normalized scenario-resource dependency graph with runtime timing annotations. Nodes represent IaC-level resources or supporting graph nodes after cleaning and normalization, while directed edges represent dependency relations. The highlighted path is the tool-side runtime-dominant dependency path for the selected replication, computed from IaC resource finish offsets. It is therefore evidence about the tool-side provisioning timeline, not a direct cloud-readiness path.

The timeline chart below each graph shows the same run from the resource-timeline perspective. The horizontal axis reports seconds since the IaC apply start. The vertical

Case	Main graph/timeline feature	Supporting repeated-run evidence	Interpretation
YCSB Azure Terraform	Compact network–compute–cache structure, but long first-run resource spans	Mean time-to-ready 800.5 s; CV 2.9%; Azure much slower than AWS and GCP for YCSB	Small graph size does not guarantee short provisioning time; provider-specific resource lifecycle behavior dominates this case
CloudSuite Azure Pulumi	Mid-sized graph with routing, ingress, VM, NIC, and disk; short total run but non-trivial stabilization	Mean time-to-ready 62.1 s; readiness-lag share 16.5%; CV 11.9%	Fastest configuration overall, but readiness lag and variability remain relevant
DeathStarBench GCP OpenTofu	Large graph with NAT, Kubernetes, node pool, ingress, database, Redis, and registry branches	Mean time-to-ready 884.9 s; registry, database, Redis, NAT, and cluster roles show long resource-level timing windows	Largest scenario is explained by several long managed-service branches rather than a single isolated resource

Table 6.3: Cross-case dependency-based interpretation.

axis lists resources observed in the IaC event stream. Each bar shows a tool-side start-to-finish interval for one resource; blue bars identify scenario-managed resources, while grey bars identify other IaC resources. These timelines are useful for identifying resource overlap, ordering, and long tool-side spans. Cloud-side readiness is interpreted through deployment time-to-ready, readiness lag, and resource-level readiness summaries rather than through the graph finish metric.

6.3.3 Graph Depth Alone Does Not Determine Time-to-Ready

The main dependency-based finding is that graph depth alone does not determine deployment time-to-ready. Table 6.3 summarizes the three selected cases and shows three different mechanisms: provider-specific lifecycle delay in a compact graph, readiness-lag sensitivity in a fast mid-sized graph, and accumulated managed-service windows in a large application-stack graph.

The table states the numerical support for each case before interpreting the mechanism. The YCSB Azure Terraform case is slow despite a compact role structure, which means

graph size alone cannot explain it. The CloudSuite Azure Pulumi case is fast but still has the largest readiness-lag share, which means short apply-side execution does not remove readiness concerns. The DeathStarBench GCP OpenTofu case is the slowest configuration, but its explanation is not a single bottleneck; several managed-service roles have long timing windows. These cases show why dependency-based artifacts are useful: they connect aggregate metrics to resource ordering, overlap, and provider-specific lifecycle behavior.

6.3.4 A Compact Graph Can Be Slow When Redis Dominates

The YCSB-inspired Azure Terraform case shows that a compact graph can still be slow when one managed resource dominates the lifecycle. This case is selected because Azure is much slower than AWS and GCP for the YCSB-inspired scenario. In the repeated-run results, Azure Terraform has a mean time-to-ready of 800.5 s, compared with 351.6 s on AWS and 385.2 s on GCP.

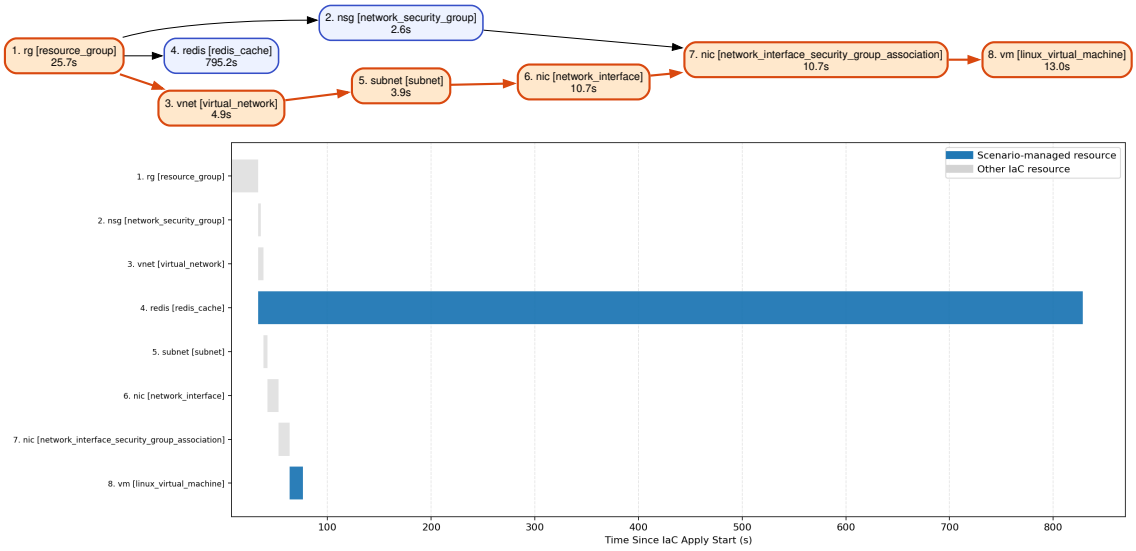

Figure 6.7: YCSB-inspired Azure Terraform first-replication dependency graph and all-resource timeline.

Figure 6.7 shows the YCSB Azure Terraform dependency graph and all-resource timeline for the first successful replication. Overall, the graph is compact and centers on the network, subnet, VM, and cache roles, while the Azure implementation also contains supporting resources such as the resource group, network security group, and network-interface

resources. In the timeline, the horizontal axis shows seconds since IaC apply start, the vertical axis lists resources, and each bar shows the tool-side span of one resource.

The visible timing pattern is dominated by the managed Redis cache span. The graph itself is not large, but the timeline shows that the Redis resource occupies most of the run. Across the five accepted replications, the case remains slow and relatively stable, with a CV of 2.9%. This means the result is unlikely to be a single anomalous run. The most plausible explanation is provider-specific managed-cache lifecycle behavior: Azure's cache resource takes much longer to reach the relevant provisioning and readiness state in this scenario than the comparable AWS and GCP YCSB configurations.

6.3.5 A Fast Run Can Still Be Readiness-Lag Sensitive

The CloudSuite-inspired Azure Pulumi case shows that a fast run can still be sensitive to post-apply stabilization. This case is selected because it is the fastest configuration overall, with a mean time-to-ready of 62.1 s, but it also has relatively high variability with a CV of 11.9%. Its readiness-lag share is the largest in the result set at 16.5%.

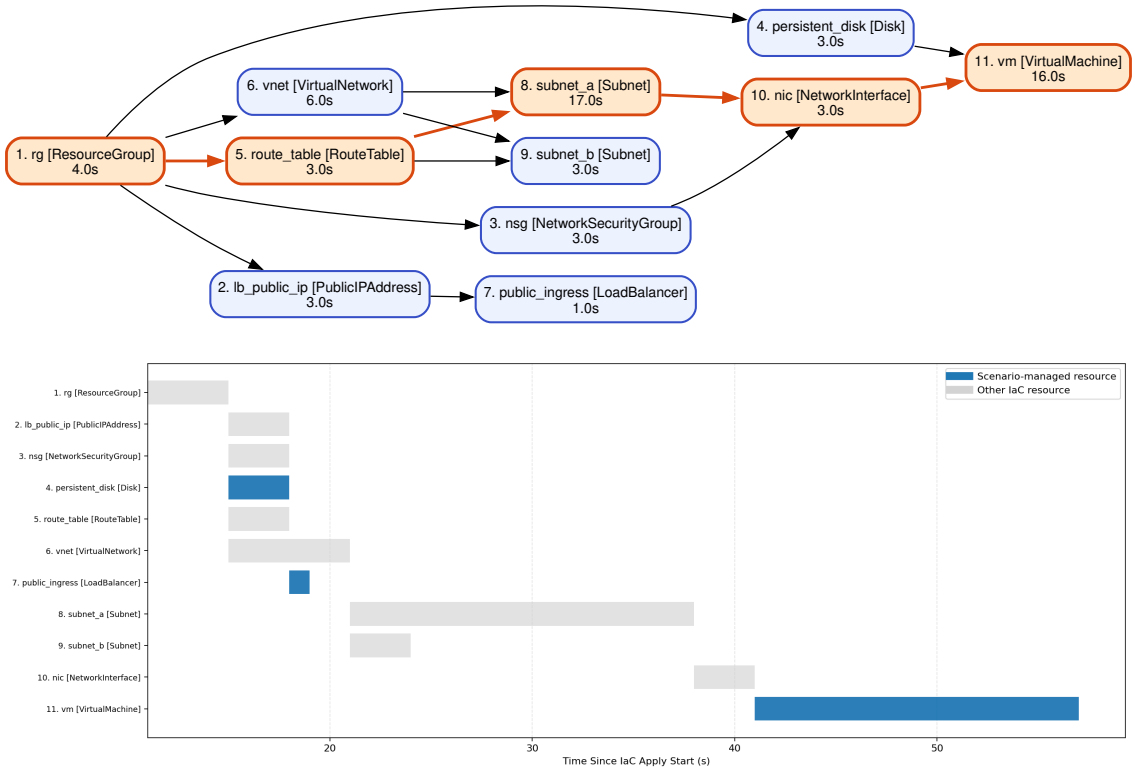

Figure 6.8: CloudSuite-inspired Azure Pulumi first-replication dependency graph and all-resource timeline.

Figure 6.8 shows a mid-sized composite graph with routing, public ingress, VM, network-interface, and persistent-disk resources. The graph shows dependencies among these resources, while the timeline shows how their tool-side spans overlap during the first successful replication. The horizontal axis again reports seconds since apply start, and the vertical axis lists the resources in the IaC event stream.

The numbers show two properties at the same time. First, the mean time-to-ready is only 62.1 s, making this the fastest configuration in the full result set. Second, readiness lag accounts for 16.5% of that total, and the CV is 11.9%. This means that the configuration is fast in absolute terms but still not negligible in post-apply behavior or relative variability. The likely explanation is that the Azure Pulumi apply phase finishes quickly for this graph, but some resource readiness predicates still complete after tool-reported apply completion. Therefore, short deployment time and readiness-lag sensitivity can coexist.

6.3.6 A Large Graph Accumulates Several Managed-Service Windows

The DeathStarBench-inspired GCP OpenTofu case shows that the largest scenario is slow because several managed-service windows accumulate across a large graph. This case is selected because it has the highest mean time-to-ready in the result set at 884.9 s. It also represents the broadest scenario structure, involving networking, NAT, managed Kubernetes, node pools, public ingress, Redis, managed database, and container registry roles.

Figure 6.9 shows that the selected run is not simply a longer version of the smaller scenarios. Overall, the graph contains multiple long branches and a deeper chain through NAT, Kubernetes, node-pool, and ingress-related resources. The timeline shows which resources occupy long tool-side spans and which resources overlap. Its horizontal axis reports seconds since apply start, its vertical axis lists resources, and each bar shows a resource's tool-side provisioning span.

The repeated-run summaries explain why this scenario remains slow even though not every long resource is on the highlighted dependency path. In GCP, the container registry, managed database, Redis, NAT, and Kubernetes cluster all show large cloud request-to-ready or resource-level readiness-lag values. The mean time-to-ready of 884.9 s therefore reflects several long resource windows across a large dependency graph rather than one isolated resource. A likely explanation is that the DeathStarBench-inspired scenario combines multiple managed services whose provider-side provisioning and stabilization proceed partly in parallel but still extend the overall lifecycle when they appear on or near important dependency branches.

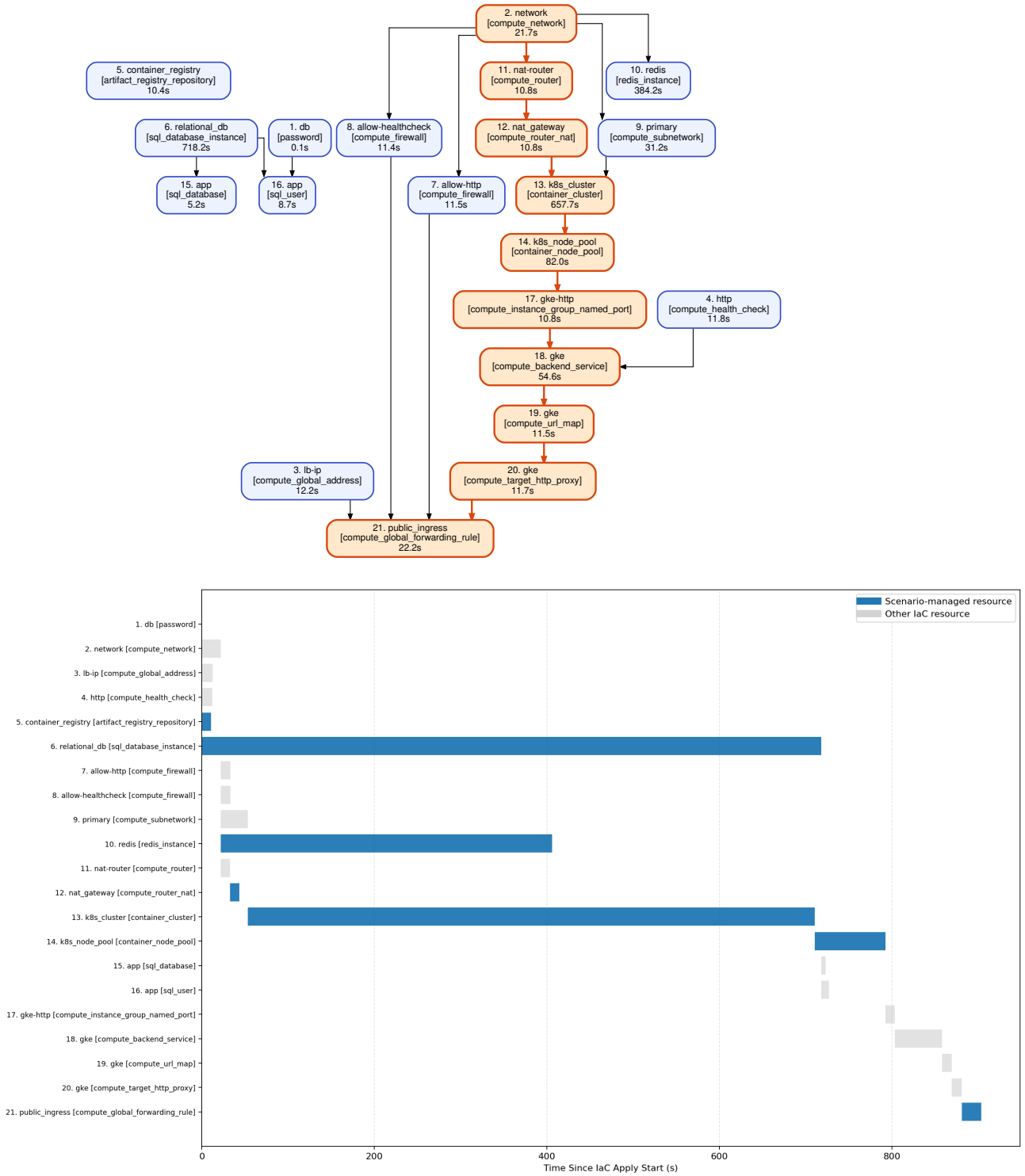

Figure 6.9: DeathStarBench-inspired GCP OpenTofu first-replication dependency graph and all-resource timeline. The dependency graph is re-arranged into a compact layout to improve readability while preserving the same resource nodes, dependency edges, timing annotations, and runtime-dominant dependency-path highlighting.

6.4 Answer to RQ3

The two experiments answer RQ3 by separating comparison from explanation. Experiment 1 compares deployment time, readiness behavior, and run-to-run variability across the full benchmark matrix. Experiment 2 explains selected outcomes through dependency graphs, tool-side resource timelines, and runtime-dominant dependency paths. Together, the results show that IaC provisioning behavior is shaped more strongly by scenario and provider effects than by a globally dominant IaC tool, and that dependency-based artifacts are necessary to explain why those aggregate differences occur.

6.4.1 Deployment-Time Findings

For deployment time, the main finding is that scenario and provider effects are stronger than universal tool effects. CloudSuite-inspired configurations are generally fastest, DeathStarBench-inspired configurations are consistently slowest, and YCSB-inspired configurations are intermediate but become much slower on Azure. Tool differences exist in individual cells, but no IaC tool is globally fastest across all providers and scenarios. This means that tool choice should be interpreted together with provider and scenario context rather than as an isolated ranking.

6.4.2 Readiness-Behavior Findings

For readiness behavior, the evaluation confirms that tool-reported completion and cloud-observed usability should be treated as separate lifecycle events. Apply time dominates most end-to-end measurements, but readiness lag remains important in selected cases, especially where a short deployment still has a visible post-apply stabilization share. Resource-level summaries further show that readiness behavior is role- and provider-dependent: VM, persistent disk, NAT, registry, database, Redis, and Kubernetes-related resources can all become important depending on the scenario and provider.

6.4.3 Variability Findings

For run-to-run variability, the results show that relative stability is not determined by mean time alone. Some slow configurations are relatively stable, while some fast configurations are relatively variable. This is why the evaluation reports coefficient of variation in addition to mean time-to-ready: it separates configurations that are slow but predictable from configurations that are fast but less stable.

6.4.4 Dependency-Based Explanation Findings

The dependency-based artifacts explain why aggregate timing results differ. They show resource ordering, overlap, long tool-side spans, and runtime-dominant dependency paths in representative runs. The key explanation is that structural depth alone is insufficient: a compact graph can still be slow when a provider-specific managed resource dominates timing, a fast mid-sized graph can still show readiness-lag sensitivity, and a large application-stack graph can accumulate long timing windows across multiple managed-service branches. Thus, dependency graphs and timelines do not replace the repeated-run statistics; they provide the structural explanation needed to interpret them.

6.5 Threats to Validity

These findings should be interpreted within the measurement and artifact boundaries defined earlier in the thesis. In particular, the repeated-run summaries support quantitative comparison, while the graph and timeline artifacts support run-level structural explanation. The following threats to validity clarify the limits of this interpretation.

The evaluation is designed to make provisioning behavior comparable across tools, providers, and scenarios, but the results still depend on several methodological and operational choices. These threats mainly concern whether the selected metrics capture the intended concepts, whether uncontrolled execution conditions could affect the observed differences, how far the selected scenarios generalize, and how first-run graph visualizations should be interpreted. The following subsections discuss these limitations and define the boundary within which the results should be read.

6.5.1 Construct Validity

The main construct-validity threat is whether the selected metrics capture the intended provisioning concepts. Deployment time-to-ready measures the interval from IaC apply start to externally observed readiness, which matches the lifecycle view of this thesis. Apply time captures tool-side execution, while readiness lag captures post-apply stabilization. However, runtime-dominant dependency paths in the current dependency-based analysis are computed from tool-side resource spans rather than cloud-side readiness offsets. This choice preserves fine-grained alignment with the IaC-derived graph nodes, because cloud-side readiness observations are not available at the same graph-node granularity. The

consequence is that post-apply readiness delays are interpreted through lifecycle metrics rather than directly through the graph-highlighted path.

6.5.2 Internal Validity

Internal validity depends on whether the observed differences are caused by the evaluated tool-provider-scenario configurations rather than by uncontrolled execution conditions. The experiment controls the benchmark matrix, resource roles, regions, repeated-run procedure, naming, cleanup, and final data inclusion rules. Still, cloud provisioning is not fully deterministic. Provider-side scheduling, API latency, temporary capacity conditions, quota behavior, and managed-service queues may affect individual runs. Repeating each configuration five times reduces the risk of relying on one anomalous run, but it does not eliminate cloud nondeterminism.

6.5.3 External Validity

The evaluation covers three IaC tools, three major cloud providers, and three benchmark-inspired infrastructure scenarios. This gives broad coverage for the thesis scope, but it does not represent all IaC tools, all providers, all regions, or all possible infrastructure designs. The scenarios are infrastructure abstractions inspired by benchmark families, not complete application workload deployments. Therefore, the results should be generalized to IaC provisioning behavior for similar resource-graph scenarios, not to application-level workload performance.

6.5.4 First-Run Visualization Limitation

Experiment 2 uses graph and timeline artifacts from the first successful replication of the selected configurations. These artifacts are useful for explaining resource ordering, overlap, and highlighted dependency paths, but they are not median-run or average-run visualizations. For this reason, the figures are interpreted together with repeated-run summary statistics. Claims about typical performance and variability are based on the five-run summary data, not on a single visual artifact alone.

6.5.5 Cloud Nondeterminism

Cloud nondeterminism remains an unavoidable threat in this evaluation. Managed services may be queued, initialized, or stabilized differently across repeated runs. Readiness checks may also observe provider state at slightly different moments depending on polling and

API response behavior. These effects are part of real cloud provisioning behavior, but they make exact reproduction of individual run timings difficult. The evaluation therefore emphasizes patterns across repeated runs and uses dependency-based artifacts to explain selected cases rather than to claim deterministic execution schedules.

7

Related Work

This chapter positions the thesis against prior work on Infrastructure-as-Code (IaC), cloud benchmarking, and adjacent benchmarking methodologies. The goal is not only to summarize existing studies, but also to clarify how this thesis differs from work that studies IaC artifacts, uses IaC to automate cloud experiments, benchmarks cloud-provider resources, or evaluates application-level workloads. The chapter first reviews these related areas separately, then synthesizes the remaining gap: prior work provides important ingredients for reproducible cloud benchmarking, but does not integrate them into a lifecycle-aware and dependency-based benchmark of IaC provisioning behavior.

7.1 Infrastructure-as-Code Research and Tool-Level Reasoning

Research on IaC has established that infrastructure definitions can be treated as software artifacts and studied using software-engineering methods. Rahman et al. provide a systematic mapping study of IaC research, showing that IaC has been studied in relation to testing, quality, defects, security, and software-engineering practice (8). Kumara et al. complement this view through a systematic gray-literature review of infrastructure code practices, identifying recurring recommendations and anti-patterns used by practitioners (7). These works are relevant because they motivate IaC as a research object rather than merely as an operational convenience.

This thesis builds on that view, but focuses on a different aspect of IaC. Much IaC research studies the quality, maintainability, correctness, or practice of infrastructure code. In contrast, this thesis studies how IaC tools behave when they provision cloud resource graphs. The object of analysis is therefore not only the static IaC definition, but also the

provisioning lifecycle that unfolds when the definition is applied across tools, providers, and infrastructure scenarios.

A closer performance-oriented study is the work by Wang et al., which discusses IaC and assesses performance and availability on Google Cloud Platform (10). Its connection to this thesis is empirical: it moves beyond IaC concepts and studies observed behavior. Its scope is narrower, however, because it does not provide a cross-tool and cross-cloud benchmark of IaC provisioning behavior, nor does it connect runtime observations to normalized dependency graphs, runtime-dominant dependency paths, or fine-grained provisioning timelines.

This positions prior IaC research as important motivation, but not as a direct solution to the problem studied in this thesis. The remaining gap is the lack of a method that treats IaC provisioning as a dependency-structured runtime process. This thesis therefore shifts the focus from IaC artifacts and practices to the graph-structured provisioning behavior that emerges when IaC tools create cloud infrastructure.

7.2 IaC-Enabled Cloud Benchmarking and Experiment Automation

The closest related work in cloud experiment automation is Cloud WorkBench, which uses IaC to define and execute reproducible cloud benchmarks (3). It demonstrates that infrastructure definitions can make cloud benchmarking more repeatable and configurable. In Cloud WorkBench, however, IaC is mainly an enabling mechanism for running benchmarks: the system under test is the cloud configuration or resource behavior, not the IaC provisioning tool itself.

Recent work such as Multi-IaC-Eval also shows growing interest in benchmarking IaC artifacts directly. Multi-IaC-Eval introduces Multi-IaC-Bench, a benchmark dataset for evaluating LLM-based IaC generation and mutation across AWS CloudFormation, Terraform, and Cloud Development Kit (CDK) formats (31). The connection to this thesis is that IaC representations are treated as evaluation objects rather than only as scripts used to launch other experiments. The difference is that Multi-IaC-Eval studies format-level IaC generation and modification by language models, whereas this thesis studies repeated real-cloud provisioning by IaC tools.

Smart CloudBench is another relevant example of cloud benchmark automation. It provides a framework for evaluating cloud infrastructure performance and supports systematic

benchmarking across cloud environments (13). This line of work is relevant because it emphasizes controlled execution, automation, and repeatability in cloud benchmarking, which are also important for the standardized benchmark lifecycle used in this thesis.

These works show that IaC and automation are valuable for reproducible cloud experiments, but they do not address the specific lifecycle-performance problem studied here. They do not separate tool-observed completion from cloud-observed readiness, and they do not use dependency graphs as the explanatory structure for provisioning behavior. In contrast, this thesis treats IaC provisioning as the system under test: it executes real deployments, records tool-side and cloud-side lifecycle observations, and links those observations to normalized resource dependency graphs.

7.3 Cloud-Provider and Cloud-Resource Benchmarking

Cloud-provider benchmarking provides the broader methodological background for this thesis. CloudCmp is a classic example of systematic public-cloud comparison, benchmarking multiple cloud providers using defined workloads and measurement procedures (2). Iosup et al. further discuss approaches, challenges, and experience in IaaS cloud benchmarking, emphasizing the difficulty of designing fair and meaningful comparisons in cloud environments (11). These works are relevant because this thesis also compares behavior across cloud providers and must account for provider-specific resource models, timing behavior, and variability.

Prior cloud benchmarking work also motivates the need for repeated measurements and careful interpretation of variability. Laaber et al. show that software microbenchmarking in the cloud can be affected by substantial variability and that cloud measurement requires careful experimental design (9). More generally, cloud computing exposes configurable and elastic infrastructure resources whose performance and behavior depend on provider implementation details and operational conditions (1). These observations support the repeated-run and variability-aware design used in this thesis.

The boundary of these benchmarks is different from the one used in this thesis. Cloud-provider and cloud-resource benchmarks usually study the performance of resources after they exist, or the performance of workloads running on those resources. They therefore do not primarily study the IaC provisioning lifecycle that creates the resources, distinguish IaC tool completion from later cloud-side readiness, or explain deployment behavior through IaC dependency graphs.

This thesis keeps the cross-cloud comparison perspective of cloud benchmarking, but applies it to a different system boundary. The system under test is not only the cloud resource or application workload, but the provisioning process through which IaC tools create dependency-rich infrastructure. This makes the benchmark lifecycle extend from IaC apply start, through tool-reported completion, to cloud-observed readiness. The resulting analysis connects provider differences to dependency structure, readiness predicates, and runtime-dominant dependency paths.

7.4 Benchmark-Inspired Workloads and Adjacent Benchmarking Methodologies

The infrastructure scenarios in this thesis are inspired by established benchmark families, but they are not intended to reproduce those benchmarks at the application-workload level. YCSB is a benchmark for cloud serving systems and motivates the data-serving scenario used in this thesis (32). CloudSuite studies emerging scale-out workloads and motivates the composite cloud-service scenario (4). DeathStarBench provides a benchmark suite for microservices and their cloud and edge system implications, motivating the application-stack scenario (5).

These benchmark families are relevant because they provide recognizable workload classes and infrastructure patterns: data-serving systems, scale-out service deployments, and microservice-oriented application stacks. In this thesis, they are used as scenario motivation rather than as application workloads. The scenarios therefore abstract the infrastructure roles needed to provision each workload class, while leaving datastore throughput, scale-out workload performance, and microservice request latency outside the scope of the evaluation.

Adjacent benchmarking work also informs the structure of this thesis. Straesser et al. propose a systematic approach for benchmarking container orchestration frameworks (12), which is methodologically close to this thesis because it benchmarks a cloud-management layer and makes benchmark scope, requirements, metrics, and methodology explicit. Established measurement-methodology work similarly motivates the careful connection between goals, questions, and metrics in empirical software-engineering studies (33).

However, these adjacent benchmarks target different layers of the cloud software stack. YCSB, CloudSuite, and DeathStarBench focus on workload or application behavior, while container-orchestration benchmarking focuses on orchestration frameworks. They do not benchmark IaC tools that provision cloud resource graphs, and they do not analyze the

lifecycle from IaC apply to cloud readiness. This thesis therefore uses established benchmark families only as inspiration for scenario design, while shifting the benchmark target to IaC provisioning behavior.

7.5 Synthesis: Gap and Novelty of This Thesis

The related work reviewed above provides several foundations for this thesis. IaC research establishes infrastructure code as a meaningful software-engineering object of study. IaC-enabled benchmarking shows that infrastructure definitions can improve reproducibility and automation in cloud experiments. Cloud-provider benchmarking provides methodological guidance for repeated and controlled measurement in variable cloud environments. Established benchmark suites motivate representative scenario classes, while adjacent benchmarking work shows the value of explicit benchmark scope, metrics, and architecture.

The remaining gap lies in how these foundations are combined. Prior work does not provide an integrated benchmark in which IaC provisioning itself is the system under test, the provisioning lifecycle is measured beyond tool-reported completion, and the resulting timing behavior is explained through dependency-graph structure. In particular, existing work does not jointly combine cross-tool execution, cross-cloud comparison, cloud-observed readiness, readiness-lag measurement, normalized dependency graphs, runtime-dominant dependency paths, and repeated-run variability analysis.

This thesis addresses that gap by combining three elements in one framework. First, benchmark-inspired infrastructure scenarios are represented as dependency graphs so that provisioning structure can be compared across tools and cloud providers. Second, the benchmark framework measures the lifecycle from IaC apply start to cloud-observed readiness, rather than stopping at tool completion. Third, the evaluation links repeated timing results to graph and timeline artifacts, allowing provisioning behavior to be compared and explained. The novelty of the thesis is therefore the integration of lifecycle-aware measurement and dependency-based explanation for IaC provisioning across tools, providers, and infrastructure scenarios.

8

Conclusion

This thesis studied Infrastructure-as-Code (IaC) provisioning as a measurable cloud lifecycle rather than only as a configuration or automation practice. The work was motivated by the observation that reproducible infrastructure definitions do not by themselves guarantee predictable provisioning behavior. Even when the same infrastructure scenario can be expressed and rerun with IaC, the time until resources become usable may differ across tools, cloud providers, resource types, and repeated executions. For this reason, the thesis argued that IaC provisioning should be evaluated through both lifecycle-aware measurement and dependency-based explanation.

To address this problem, the thesis made three contributions. First, it defined a dependency-graph analysis method for IaC provisioning. This method represents provisioning scenarios as resource dependency graphs and normalizes tool-specific graph artifacts into scenario-resource graphs. It supports structural analysis through graph depth, parallelizable layers, dependency paths, and structural dependency-path candidates, and supports runtime explanation through runtime-dominant dependency paths. Second, it designed and implemented a lifecycle-aware benchmarking framework that executes IaC scenarios under a standardized workflow, captures tool-side apply timing, observes cloud-side readiness, and stores structured artifacts for end-to-end and dependency-based analysis. Third, it evaluated Terraform, OpenTofu, and Pulumi across AWS, Azure, and GCP using three benchmark-inspired infrastructure scenarios and repeated runs.

8.1 Research Answers and Main Findings

The first research question asked how IaC provisioning should be analyzed in terms of resource dependencies. This thesis answered it by showing that dependency graphs pro-

vide a useful common representation for comparing provisioning structure across tools and providers. The graph abstraction makes resource dependencies explicit, separates scenario-relevant resources from tool-internal artifacts, and allows runtime observations to be attached to graph nodes. This makes it possible to interpret provisioning behavior not only as an aggregate duration, but also as a process shaped by resource roles, dependency chains, parallelizable layers, and runtime-dominant paths.

The second research question asked how a lifecycle-aware benchmarking framework should be designed. The framework developed in this thesis answers this question through a manifest-driven and artifact-oriented design. Each run follows the same high-level lifecycle: configuration loading, environment setup, tool initialization, apply execution, tool-side timeline capture, output resolution, scenario-resource resolution, cloud observation, metric computation, cleanup, and result storage. Tool-specific drivers and provider-specific observers allow Terraform, OpenTofu, Pulumi, AWS, Azure, and GCP to be handled through a common benchmark structure without assuming that their implementations are identical. This design supports fair repeated comparison while preserving the information needed for later dependency-based explanation.

The third research question asked how IaC tools compare across providers and scenarios, and what dependency graphs and timelines reveal about the causes of observed differences. The evaluation showed that scenario and provider effects are stronger than a globally dominant IaC tool effect. Across the 27 tool-provider-scenario configurations and 135 successful runs, no single IaC tool was consistently fastest across all cases. Tool choice still mattered in specific configurations, but provider behavior and scenario composition had larger effects on overall time-to-ready.

The results also showed that provisioning behavior is scenario-sensitive. The CloudSuite-inspired scenario was generally the fastest, the DeathStarBench-inspired scenario was consistently the slowest, and the YCSB-inspired scenario showed strong provider-specific behavior, especially on Azure. Provider behavior did not form a single global ranking: a provider that was fast for one scenario was not necessarily fastest for another. This supports the need for benchmarking IaC provisioning across scenario classes rather than drawing conclusions from a single infrastructure example.

The lifecycle-aware measurements further showed that tool-observed completion and cloud-observed readiness should be treated as separate events. In most configurations, IaC apply time dominated deployment time-to-ready, but readiness lag still mattered in selected cases. The CloudSuite-inspired Azure Pulumi configuration was the fastest overall, yet still had a visible readiness-lag share and relatively high relative variability. This illustrates

why stopping measurement at tool completion can miss part of the practical deployment lifecycle.

Finally, the dependency-based analysis showed that aggregate timing results need structural explanation. In the YCSB-inspired Azure Terraform case, the graph was compact, but the managed Redis cache dominated the runtime timeline, showing that small graphs can still produce long provisioning times because of provider-specific managed-service behavior. In the CloudSuite-inspired Azure Pulumi case, the configuration was fast overall, but lifecycle decomposition revealed non-trivial post-apply stabilization. In the DeathStarBench-inspired GCP OpenTofu case, the large application-stack graph exposed several managed-service and orchestration branches, showing how complex scenarios accumulate long provisioning windows across multiple resource roles. These cases demonstrate that dependency graphs and timelines help explain why end-to-end results differ.

8.2 Limitations and Future Work

The scope of this thesis is intentionally bounded. The evaluation considered three IaC tools, three cloud providers, three benchmark-inspired scenarios, and five successful replications per configuration. This scope was sufficient to demonstrate the method, framework, and main empirical patterns, but it does not cover all IaC tools, providers, regions, resource types, or operational settings. Future work should extend the framework to additional tools such as CloudFormation, CDK-based systems, Crossplane, or Ansible-style workflows, and should evaluate additional cloud regions and scenario classes.

The cross-cloud comparison also relies on role-level equivalence rather than identical resources. This is necessary because AWS, Azure, and GCP expose different services and readiness semantics. However, role equivalence does not mean behavioral equivalence. Future work could refine the scenario abstraction by distinguishing between basic role equivalence, feature-level equivalence, and stricter performance-oriented equivalence.

Another limitation is that the current runtime-dominant dependency-path analysis uses tool-side resource spans as the graph-level timing source. This is appropriate because those spans align directly with IaC-derived graph nodes, but cloud-side readiness observations are currently used mainly for deployment time-to-ready, readiness lag, and lifecycle interpretation. Future work should improve fine-grained alignment between cloud-side readiness observations and graph nodes, so that runtime-dominant dependency paths can incorporate both tool completion and cloud readiness.

The benchmark also focuses on provisioning from a clean scenario state. Each measured run creates the selected scenario resources, observes readiness, and then destroys those resources for cleanup. Real IaC workflows may also start from resources that already exist, import unmanaged resources into IaC state, modify an already deployed environment, or repair differences between the deployed infrastructure and the desired configuration. These workflows are outside the scope of this thesis. Although cleanup is recorded for robustness, destroy-time behavior is not analyzed as a first-class benchmark target. Extending the framework to existing-resource, update, repair, and destroy-focused workflows would make the lifecycle model more complete.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. **A view of cloud computing**. *Commun. ACM*, **53**(4):50–58, 2010. 1, 17, 127

[2] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. **CloudCmp: comparing public cloud providers**. In Mark Allman, editor, *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 1–14. ACM, 2010. 1, 2, 3, 14, 15, 17, 59, 127

[3] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald C. Gall. **Cloud WorkBench - Infrastructure-as-Code Based Cloud Benchmarking**. *CoRR*, **abs/1408.4565**, 2014. 1, 2, 12, 15, 59, 126

[4] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. **Clearing the clouds: a study of emerging scale-out workloads on modern hardware**. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 37–48. ACM, 2012. 1, 18, 128

[5] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. **An Open-Source Benchmark Suite for Microservices and Their**

Hardware-Software Implications for Cloud & Edge Systems. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 3–18. ACM, 2019. 1, 18, 128

[6] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. **FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices**. *CoRR*, **abs/2008.08509**, 2020. 1, 18

[7] Indika Kumara, Martin Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. **The do's and don'ts of infrastructure code: A systematic gray literature review**. *Inf. Softw. Technol.*, **137**:106593, 2021. 1, 2, 3, 11, 12, 125

[8] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie A. Williams. **A systematic mapping study of infrastructure as code research**. *Inf. Softw. Technol.*, **108**:65–77, 2019. 1, 2, 3, 11, 12, 125

[9] Christoph Laaber, Joel Scheuner, and Philipp Leitner. **Software microbenchmarking in the cloud. How bad is it really?** *Empir. Softw. Eng.*, **24**(4):2469–2508, 2019. 1, 3, 14, 15, 59, 127

[10] Hongyu Wang, Brian Kishiyama, David Lopez, and Jeong Yang. **An Overview of Infrastructure as Code (IaC) with Performance and Availability Assessment on Google Cloud Platform**. In Kevin Daimi and Abeer Alsadoon, editors, *Proceedings of the Second International Conference on Advances in Computing Research, ACR 2024, Madrid, Spain, June 3-5, 2024*, Lecture Notes in Networks and Systems, pages 497–514. Springer, 2024. 1, 2, 3, 15, 126

[11] Alexandru Iosup, Radu Prodan, and Dick H. J. Epema. **IaaS Cloud Benchmarking: Approaches, Challenges, and Experience**. In Xiaolin Li and Judy Qiu, editors, *Cloud Computing for Data-Intensive Applications*, pages 83–104. Springer, 2014. 1, 2, 3, 15, 17, 59, 127

[12] Martin Straesser, Jonas Mathiasch, André Bauer, and Samuel Kounev. **A Systematic Approach for Benchmarking of Container Orchestration**

Frameworks. In Marco Vieira, Valeria Cardellini, Antinisca Di Marco, and Petr Tuma, editors, *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15-19, 2023*, pages 187–198. ACM, 2023. 1, 2, 3, 128

[13] Mohan Baruwal Chhetri, Sergei Chichin, Quoc Bao Vo, and Ryszard Kowalczyk. **Smart CloudBench - A framework for evaluating cloud infrastructure performance**. *Inf. Syst. Frontiers*, **18**(3):413–428, 2016. 2, 127

[14] HashiCorp. **What is Terraform?** `https://developer.hashicorp.com/terraform/intro`. 12

[15] OpenTofu. **OpenTofu: Open-Source Infrastructure as Code**. `https://opentofu.org/docs/`. 12, 15

[16] HashiCorp. **Dependency Graph**. `https://developer.hashicorp.com/terraform/internals/graph`. 12, 15, 17, 23, 39, 42, 54

[17] Pulumi. **Pulumi Infrastructure as Code Documentation**. `https://www.pulumi.com/docs/iac/`. 12

[18] HashiCorp. **terraform apply command reference**. `https://developer.hashicorp.com/terraform/cli/commands/apply`. 14, 70

[19] OpenTofu. **Command: apply**. `https://opentofu.org/docs/cli/commands/apply/`. 14, 70

[20] Pulumi. **pulumi up**. `https://www.pulumi.com/docs/iac/cli/commands/pulumi_up/`. 14, 70

[21] HashiCorp. **Providers**. `https://developer.hashicorp.com/terraform/language/providers`. 15

[22] Amazon Web Services. **Status checks for Amazon EC2 instances**. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring-system-instance-status-check.html`. 15

[23] Amazon Web Services. **Health checks for Application Load Balancer target groups**. `https://docs.aws.amazon.com/elasticloadbalancing/latest/application/target-group-health-checks.html`. 15

[24] Amazon Web Services. **Viewing Amazon RDS DB instance status**. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/accessing-monitoring.html. 15

[25] Amazon Web Services. **cache-cluster-available**. https://docs.aws.amazon.com/cli/latest/reference/elasticache/wait/cache-cluster-available.html. 15

[26] Kubernetes. **Node Status**. https://kubernetes.io/docs/reference/node/node-status/. 15

[27] HashiCorp. **Terraform Registry: Providers**. https://registry.terraform.io/browse/providers. 15

[28] OpenTofu. **Resource Graph**. https://opentofu.org/docs/internals/graph/. 17, 23, 39, 42, 54

[29] Pulumi. **dependsOn: Resource Option**. https://www.pulumi.com/docs/iac/concepts/resources/options/dependson/. 17, 23, 39, 42, 54

[30] Arthur B. Kahn. **Topological sorting of large networks**. *Commun. ACM*, **5**(11):558–562, 1962. 47

[31] Sam Davidson, Li Sun, Bhavana Bhasker, Laurent Callot, and Anoop Deoras. **Multi-IaC-Eval: Benchmarking Cloud Infrastructure as Code Across Multiple Formats**. *CoRR*, **abs/2509.05303**, 2025. 126

[32] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. **Benchmarking cloud serving systems with YCSB**. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010. 128

[33] Victor R. Basili and David M. Weiss. **A Methodology for Collecting Valid Software Engineering Data**. *IEEE Trans. Software Eng.*, **10**(6):728–738, 1984. 128

Appendix A

Reproducibility

This appendix documents the reproducibility material for the framework and evaluation presented in this thesis. It complements Chapter 4, which describes the framework design, and Chapter 5, which defines the experimental setup. The purpose of this appendix is not to repeat the full repository manual, but to summarize the artifact scope, environment assumptions, execution workflow, generated outputs, and reproducibility limitations needed to inspect, rerun, and extend the benchmark.

The reproducibility material supports two levels of reproduction. First, the analysis can be reproduced from the provided result bundles without rerunning cloud provisioning. This level is sufficient to regenerate or inspect the summary, ranking, timeline, and dependency-graph artifacts used in Chapter 6. Second, the full benchmark workflow can be rerun from the experiment manifests when suitable cloud credentials, quotas, provider permissions, and IaC tools are available. The second level is more expensive and less deterministic because the benchmark provisions real cloud resources whose timing depends on provider-side conditions.

A.1 Artifact Scope

The artifact consists of the framework repository, the experiment definitions, the IaC templates, and the generated result bundles. The framework executes IaC provisioning scenarios across Terraform, OpenTofu, and Pulumi; observes cloud-side readiness on AWS, Azure, and GCP; and produces the summary, timeline, and dependency-graph artifacts used for analysis.

The repository uses short scenario names in paths and scripts. In the thesis, these correspond to benchmark-inspired scenario names:

Repository name	Thesis name	Description
`level1`	YCSB-inspired data-serving scenario	Compact data-serving infrastructure with network, subnet, VM, and managed cache roles.
`level2`	CloudSuite-inspired composite service scenario	Intermediate composite-service infrastructure with networking, VM, disk, and public ingress roles.
`level3`	DeathStarBench-inspired application-stack scenario	Larger application-stack infrastructure with networking, NAT, Kubernetes, node pool, ingress, database, cache, and registry roles.

Table A.1: Mapping between repository scenario names and thesis scenario names.

The artifact includes three main categories of outputs:

1. **Per-run artifacts.** These include copied experiment manifests, effective variables, IaC logs, resolved outputs, per-replication results, cloud observations, cleanup status, and observation-quality information.
2. **Aggregated summaries.** These include one summary JSON file per tool-provider-scenario configuration. They are the primary source for end-to-end timing, readiness-lag, and variability analysis.
3. **Post-run analysis artifacts.** These include all-resource timeline charts, timeline CSV files, normalized dependency graphs, graph sidecar files, and graph metadata used for dependency-based explanation.

The final thesis evaluation uses 27 tool-provider-scenario configurations, with five successful replications per configuration. This gives 135 accepted provisioning runs for the repeated-run analysis. A repository snapshot may contain only a subset of generated artifacts; the complete thesis evaluation should be interpreted against the final result bundles used for Chapter 6.

A.2 Repository Layout

The most relevant repository directories are summarized below. Generated output directories such as `runs/`, `summary_bundle/`, `timeline_analysis/`, and `graphs/` are normally produced by the workflow rather than edited manually.

`src/iac_bench/cli/` Command-line entry points.

`src/iac_bench/framework/` Experiment orchestration, summary generation, graph export, and timeline analysis.

`src/iac_bench/cloud/` Cloud-side observers and readiness checks.

`src/iac_bench/iac/` IaC tool drivers for Terraform, OpenTofu, and Pulumi.

`src/iac_bench/metrics/` Timeline and event parsing helpers.

`src/iac_bench/scenarios/` Scenario resolution logic.

`configs/experiments/` Official experiment manifests for the 27 tool-provider-scenario configurations.

`scenarios/` Scenario specification files used to resolve and observe scenario resources.

`iac_templates/` Terraform/OpenTofu and Pulumi IaC templates.

`scripts/` Helper scripts for official runs, summary bundling, timeline generation, graph export, and artifact bundling.

`runs/` Generated per-run artifacts.

`summary_bundle/` Aggregated summary JSON files used for repeated-run analysis.

`timeline_analysis/` Generated timeline charts and machine-readable timeline files.

`graphs/` Generated dependency graph SVG, DOT, JSON, and metadata files.

A.3 Environment and Dependencies

The framework is intended to run from a development machine or experiment runner with stable network access to the selected cloud providers. The benchmark measures cloud provisioning behavior rather than local compute performance, so the runner hardware is not an experimental factor. However, the runner must be able to execute the IaC tools, store logs and generated artifacts, and maintain stable connectivity during provisioning and cleanup.

A.3.1 Software Dependencies

The framework requires Python 3.10 or newer. Python dependencies are installed from the repository root:

```
python3 -m pip install -r requirements.txt
```

The selected IaC tools must be installed and available on `PATH`:

- Terraform,
- OpenTofu, using the `tofu` command,
- Pulumi.

Pulumi experiments additionally require Node.js and npm. Graph visualization requires Graphviz when SVG graph rendering is needed. If Graphviz is unavailable, DOT and JSON graph artifacts may still be generated, but SVG rendering may fail.

A.3.2 Cloud Credentials and Provider Context

Cloud authentication must be configured before running experiments. The provided manifests expect GCP and Azure credential files at:

```
credentials/gcp-service-account.json
credentials/azure-service-principal.json
```

AWS runs use the AWS profile configured in the manifest, for example `iac-bench`. Credential files should not be committed to version control.

The official experiment setup uses the following provider locations:

- AWS: `eu-west-1`,
- Azure: `belgiumcentral`,
- GCP: `europe-west1`, with `europe-west1-b` for zonal resources where required.

Provider accounts, subscriptions, projects, IAM permissions, service quotas, billing setup, and provider-specific prerequisites must be prepared before running the full benchmark matrix.

A.4 Installation and Basic Check

After installing the dependencies and configuring cloud credentials, run commands from the repository root. If the package is not installed into the Python environment, set the module path explicitly:

```
export PYTHONPATH=src
```

The command-line interface can be checked with:

```
PYTHONPATH=src python3 -m iac_bench -h
```

A small smoke test can be run with one replication of a single manifest:

```
PYTHONPATH=src python3 -m iac_bench run \
  configs/experiments/level1_aws_terraform.yaml \
  --replications 1
```

This command should initialize the selected IaC tool, provision the `level1` AWS Terraform scenario, resolve outputs, observe readiness, compute metrics, destroy the created resources, and write a run artifact directory under `runs/`.

A.5 Experiment Workflow

The official experiment matrix is defined by the full factorial combination of three scenario levels, three cloud providers, and three IaC tools. This gives 27 official manifests under `configs/experiments/`. The thesis uses five successful replications per configuration.

A single manifest can be executed with the replication count defined in the manifest:

```
PYTHONPATH=src python3 -m iac_bench run \
  configs/experiments/level2_gcp_opentofu.yaml
```

The replication count can also be overridden from the command line:

```
PYTHONPATH=src python3 -m iac_bench run \
  configs/experiments/level2_gcp_opentofu.yaml \
  --replications 5
```

For the official thesis workflow, the helper script `scripts/run_official_levels.sh` runs `level1`, `level2`, and `level3` sequentially for one provider-tool pair:

```
PYTHONPATH=src bash scripts/run_official_levels.sh aws terraform 5
PYTHONPATH=src bash scripts/run_official_levels.sh aws opentofu 5
PYTHONPATH=src bash scripts/run_official_levels.sh aws pulumi 5
PYTHONPATH=src bash scripts/run_official_levels.sh azure terraform 5
PYTHONPATH=src bash scripts/run_official_levels.sh azure opentofu 5
PYTHONPATH=src bash scripts/run_official_levels.sh azure pulumi 5
PYTHONPATH=src bash scripts/run_official_levels.sh gcp terraform 5
PYTHONPATH=src bash scripts/run_official_levels.sh gcp opentofu 5
PYTHONPATH=src bash scripts/run_official_levels.sh gcp pulumi 5
```

The third argument is the target replication count. The framework records failed or incomplete runs, but the thesis result set includes only accepted runs that satisfy the success, readiness-observation, graph/timeline-alignment, and cleanup requirements defined in Chapter 5.

A.6 Run Outputs

Each experiment group is written under `runs/`. A typical group has the following structure:

```
runs/
  20260410-120000-level1_aws_terraform-abcdef1234/
    summary.json
    rep_01/
      experiment.yaml
      effective_vars.json
      results.json
      logs/
      iac/
    rep_02/
    ...
```

The most important files are:

`summary.json` Aggregated result for the experiment group.

`rep_XX/results.json` Detailed per-replication result, including timing metrics and observation status.

`rep_XX/experiment.yaml` Copied effective experiment manifest for the replication.

`rep_XX/effective_vars.json` Generated and resolved variables used by the replication.

`rep_XX/logs/` IaC tool logs and event logs.

`rep_XX/iac/` Copied IaC template working directory used for the replication.

The framework records apply timing, cloud-side observations, derived metrics, cleanup status, and observation-quality information. Runs with critical observation gaps are marked as failed rather than silently accepted as complete timing evidence.

A.7 Metrics and Analysis Concepts

The main metrics derived from the framework outputs are:

IaC apply time Elapsed time of the IaC apply or update operation.

Deployment time-to-ready Time from IaC apply start to the latest required cloud-observed readiness point.

Readiness lag Deployment time-to-ready minus IaC apply time.

Tool-side resource span Per-resource start offset, finish offset, and duration parsed from IaC tool events where available.

Cloud request-to-ready duration Time from provider-side request observation to cloud-observed readiness where available.

Resource-level readiness lag Time between tool-side resource completion and cloud-observed resource readiness where available.

Coefficient of variation Standard deviation divided by mean, used to compare relative run-to-run variability across configurations with different runtime scales.

The dependency-based analysis uses normalized resource dependency graphs, timeline-aligned node labels, runtime annotations, and dependency-path metadata to explain selected provisioning outcomes. The thesis refers to the run-specific dependency chain aligned with the latest observed tool-side finish offsets as the runtime-dominant dependency path. Some generated graph sidecar files and internal implementation fields may still use legacy names such as `critical_path`; in the thesis interpretation, these fields correspond to runtime-dominant dependency-path metadata when timing coverage is available.

A.8 Summary and Visualization Artifact Generation

Aggregated summary files are bundled from discovered `summary.json` files with:

```
bash scripts/bundle_summaries.sh
```

This creates:

- `summary_bundle/`,
- `summary_bundle.zip`.

The final thesis evaluation uses a `summary_bundle.zip` with one summary JSON file for each of the 27 tool-provider-scenario configurations.

The recommended workflow for timeline and dependency-graph artifacts is:

```
bash scripts/generate_visual_artifacts.sh
```

A different replication can be selected with:

```
bash scripts/generate_visual_artifacts.sh --rep 2
bash scripts/generate_visual_artifacts.sh --rep rep_03
```

The visualization workflow performs four main steps. First, it bundles one `results.json` file per experiment group into `one_rep_results_bundle.zip`. Second, it generates all-resource timeline charts under `timeline_analysis/`. Third, it exports flat resource dependency graphs under `graphs/`. Fourth, it bundles the graph artifacts into `level_graph_bundle.zip`. Temporary staging directories are removed by default unless the workflow is instructed to keep them.

The all-resource timeline charts are written as:

```
timeline_analysis/charts/<level_provider_tool>_all_resources.png
```

Matching machine-readable timeline files are written under:

```
timeline_analysis/per_experiment/<level_provider_tool>/
```

Important timeline files include `all_resource_timeline.csv`, `all_resource_numbering.json`, and `managed_resource_timeline.csv`. The all-resource numbering file is used to align dependency graph node labels with the corresponding timeline rows.

The dependency graph outputs are written as:

```
graphs/<level_provider_tool>_graph.svg
graphs/<level_provider_tool>_graph.dot
graphs/<level_provider_tool>_graph.json
graphs/<level_provider_tool>_graph_metadata.json
```

The SVG file is the visualization artifact. The DOT, JSON, and metadata sidecar files preserve machine-readable graph structure and presentation metadata.

A.9 Reproducing the Thesis Analysis

The thesis analysis can be reproduced in two ways.

A.9.1 Reproducing Analysis from Existing Result Bundles

When the final result bundles are available, cloud provisioning does not need to be rerun. The reproduction workflow is:

1. provide or unpack `summary_bundle.zip`;
2. regenerate or inspect end-to-end, variability, and resource-level summary figures from the summary JSON files;
3. provide or regenerate `timeline_analysis/` and `graphs/`;
4. inspect the representative timeline and dependency-graph artifacts used in Chapter 6;
5. compare the regenerated figures and artifacts with the thesis figures.

This path verifies the analysis workflow and figure generation without incurring cloud cost.

A.9.2 Rerunning Cloud Provisioning Experiments

When cloud credentials, quotas, and IaC tools are available, the full cloud benchmark can be rerun:

1. run each official provider-tool batch with five replications;
2. bundle summaries with `scripts/bundle_summaries.sh`;
3. generate timeline and graph artifacts with `scripts/generate_visual_artifacts.sh`;

4. regenerate or inspect the evaluation figures from the produced artifacts.

Exact timing values are not expected to match the thesis bit-for-bit. Cloud provisioning depends on provider-side scheduling, regional capacity, quota conditions, API behavior, transient delays, and account-specific limits. A successful reproduction should instead produce the same artifact structure and support the same analysis workflow.

A.10 Expected Outputs and Validation

For the full 27-configuration benchmark matrix with five accepted replications per configuration, the final analysis material should contain:

- 27 aggregated summary JSON files in `summary_bundle/`;
- 135 successful run records in the accepted result set;
- 27 all-resource timeline charts under `timeline_analysis/charts/`;
- 27 dependency graph SVG files under `graphs/`;
- matching DOT, JSON, and metadata sidecar files for the dependency graphs;
- full ranking figures and reduced main-text figures derived from the same summary data.

A reproduction should be checked at three levels. First, the expected artifact structure should be present. Second, the summary files should contain the expected configurations and accepted replications. Third, the broad qualitative findings should remain interpretable through the generated summaries, graphs, and timelines: scenario and provider effects should be evaluated against tool effects, apply time and readiness lag should be decomposed, variability should be computed from repeated runs, and dependency graphs plus timelines should remain available for selected explanatory cases.

A.11 Experiment Customization

The framework can be extended along the same dimensions used in the thesis evaluation. New experiment configurations can be added by creating manifests under `configs/experiments/`. New IaC tools require a tool driver that implements the common lifecycle responsibilities.

New providers require a cloud observer that implements the common observation abstraction and provider-specific readiness checks. New scenarios require IaC templates and scenario specification files that define resource roles, output mappings, cloud identities, and readiness predicates.

When extending the benchmark, the following consistency checks should be preserved:

- scenario roles should be defined explicitly;
- concrete provider resources should be mapped to those roles;
- tracked resources should expose identities that can be resolved from IaC outputs;
- readiness predicates should be defined before experiment execution;
- graph nodes and timeline entries should be alignable through the identity model;
- repeated runs should use the same lifecycle, cleanup policy, and inclusion rules.

A.12 Reproducibility Limitations

The artifact supports reproducibility, but it does not remove the nondeterminism of cloud provisioning. Reruns may produce different absolute timings because managed services, network resources, Kubernetes clusters, databases, caches, registries, and load-balancing components are provisioned by external cloud providers. Cloud quotas, temporary provider incidents, regional capacity, API throttling, and account-specific limits can also affect whether a configuration succeeds without adjustment.

The benchmark measures provisioning from a clean scenario state. Existing provider-level prerequisites such as accounts, credentials, regions, quotas, and installed tools are part of the execution environment, but pre-existing scenario resources are not part of the measured benchmark target. Each accepted run creates the scenario resources, observes readiness, and destroys the resources after measurement.

The dependency-based graph and timeline artifacts use one representative successful replication per configuration. These figures explain resource-level ordering and dependency structure for selected runs. Claims about typical timing and variability are based on the five-replication summary statistics rather than on a single visual artifact.

Additional supplementary material that supports the experimental setup and evaluation, but is not part of the reproducibility workflow itself, is provided in Appendix B.

Appendix B

Supplementary Evaluation Material

This appendix contains supplementary material that supports the experimental setup and evaluation chapters. Section B.1 reports the provider-specific resources used to implement the scenario roles introduced in Chapter 3 and evaluated in Chapter 5. Section B.2 provides the full ranking figures corresponding to the reduced main-text figures in Chapter 6.

B.1 Provider Resource Mapping

This appendix reports the provider-specific resources used to implement the scenario roles introduced in Chapter 3 and used as experimental factor values in Chapter 5. The mapping is role-based: resources are considered comparable when they implement the same scenario role, not because the concrete provider services are identical. The tables therefore document the concrete AWS, Azure, and GCP resource types used for reproducibility, while the main evaluation interprets results at the scenario-role level.

Scenario role	**AWS**	**Azure**	**GCP**
Virtual network	VPC	Virtual Network	VPC network
Subnet	Subnet	Subnet	Subnet
Managed Redis	ElastiCache for Redis	Azure Cache for Redis	Memorystore for Redis
VM instance	EC2 instance	Azure Virtual Machine	Compute Engine instance

Table B.1: Provider resource mapping for the YCSB-inspired data-serving scenario.

Scenario role	AWS	Azure	GCP
Virtual network	VPC	Virtual Network	VPC network
Subnet	Subnet	Subnet	Subnet
Routing	Internet Gateway and route table	Route table and public IP routing	Route
Load balancer	Application Load Balancer	Standard Load Balancer	HTTP(S) Load Balancer
VM instance	EC2 instance	Azure Virtual Machine	Compute Engine instance
Network interface	Elastic Network Interface	Network Interface	Instance network interface
VM disk	EBS volume	Managed Disk	Persistent Disk

Table B.2: Provider resource mapping for the CloudSuite-inspired composite service scenario.

Scenario role	AWS	Azure	GCP
Virtual network	VPC	Virtual Network	VPC network
Subnet	Subnet	Subnet	Subnet
Internet gateway / egress routing	Internet Gateway	Cloud egress routing	Cloud Router
NAT	NAT Gateway	NAT Gateway	Cloud NAT
Load balancer	Application Load Balancer	Standard Load Balancer	HTTP(S) Load Balancer
Managed Kubernetes cluster	EKS	AKS	GKE
Node pool	EKS node group	AKS node pool	GKE node pool
Managed Redis	ElastiCache for Redis	Azure Cache for Redis	Memorystore for Redis
Managed database	Amazon RDS	Azure Database	Cloud SQL
Container registry	ECR	Azure Container Registry	Artifact Registry

Table B.3: Provider resource mapping for the DeathStarBench-inspired application-stack scenario.

B.2 Full Evaluation Ranking Figures

This appendix provides the complete ranking figures corresponding to the reduced main-text views in Chapter 6. Figures B.1, B.2, and B.3 show the full versions of Figures 6.4, 6.5, and 6.6, respectively.

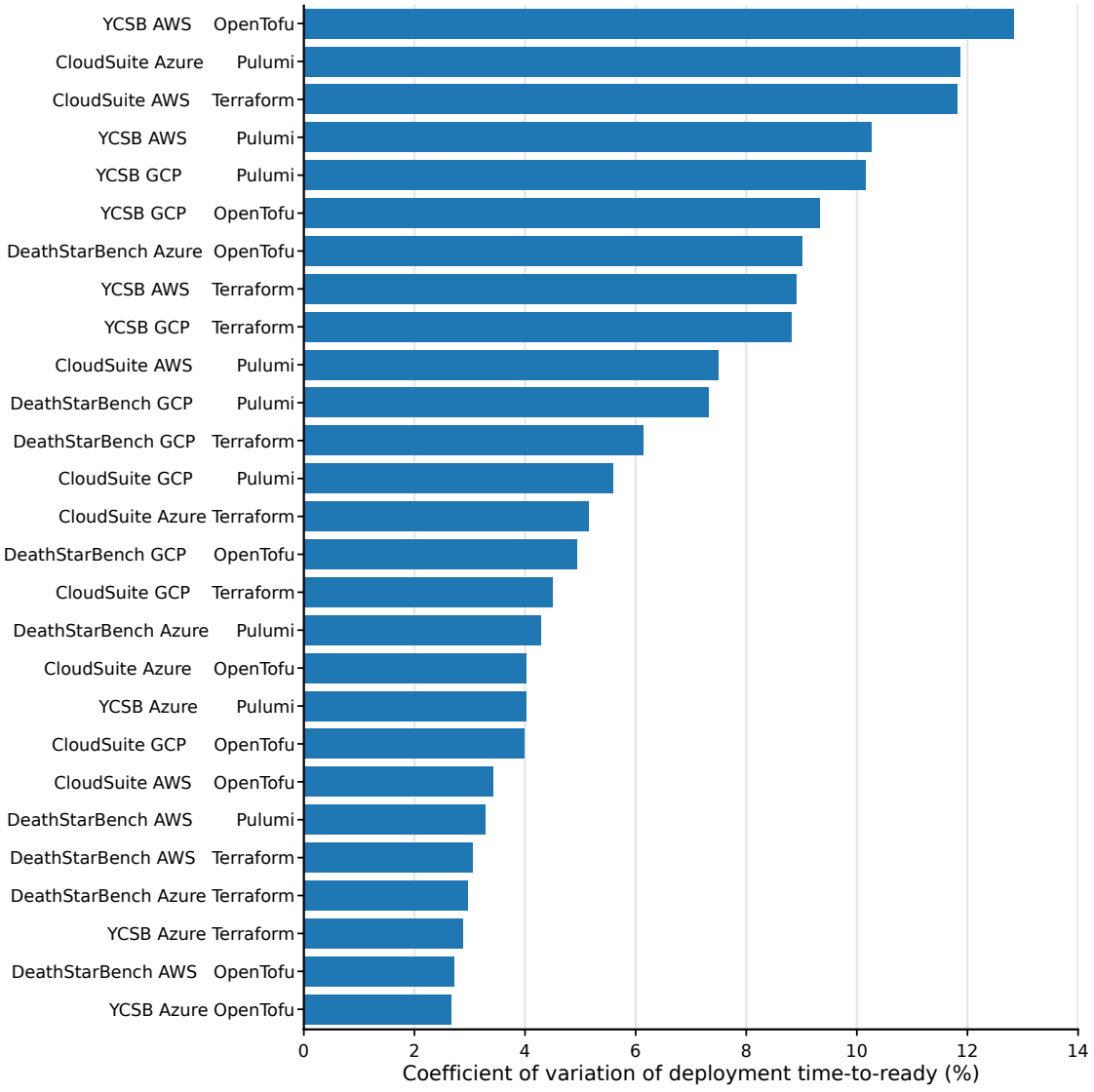

Figure B.1: Full coefficient-of-variation ranking across all tool-provider-scenario configurations. This figure is the complete version of the reduced main-text view in Figure 6.4.

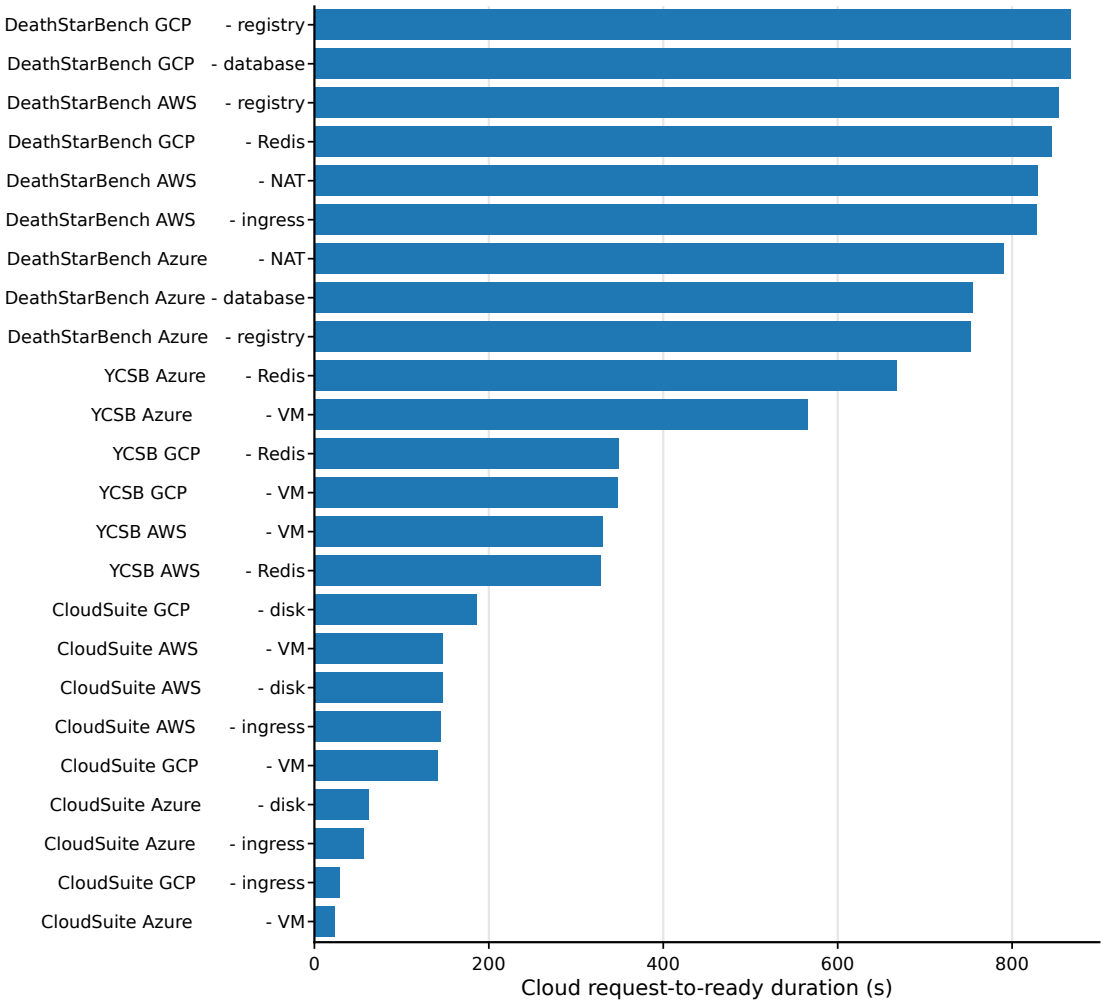

Figure B.2: Full resource-level cloud request-to-ready ranking by scenario, provider, and resource role, averaged across IaC tools. This figure is the complete version of the reduced main-text view in Figure 6.5.

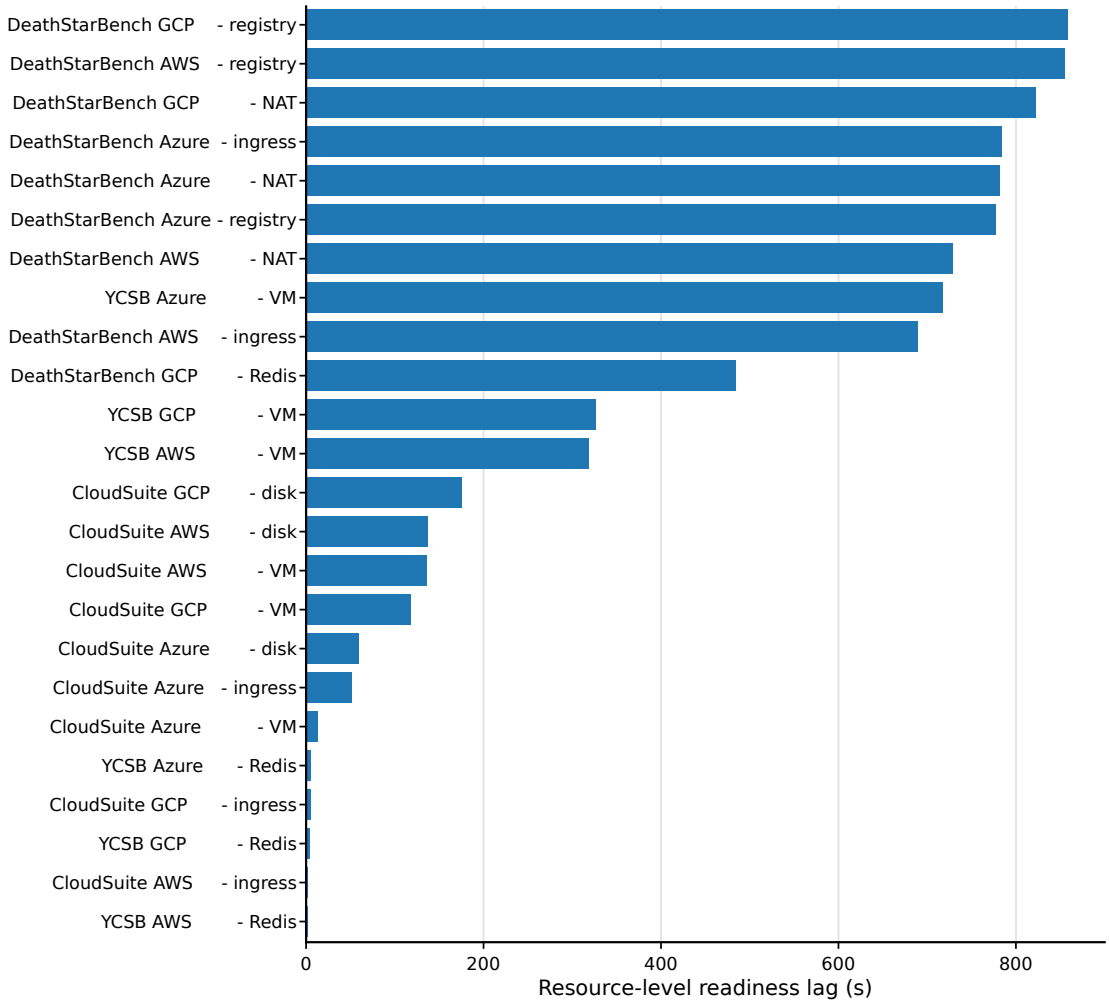

Figure B.3: Full resource-level readiness-lag ranking by scenario, provider, and resource role, averaged across IaC tools. This figure is the complete version of the reduced main-text view in Figure 6.6.