

Chapter 5

Design Objectives

The first part of this book introduced cloud service benchmarking, including its motivations and a variety of related concepts. We now focus on how to design effective cloud service benchmarks. In this chapter, we introduce the traditional key objectives of benchmark design, e.g., reproducibility, fairness, or understandability, and discuss why they are important. We then describe how these objectives need to change in the context of cloud services and cloud-based systems. Finally, we also discuss how concrete benchmarks may have to compromise one objective to reach another one and describe how the use of cloud services, both as system under test and as experimentation testbed, can influence these objectives.

5.1 Requirements for Benchmarks

Deciding whether a particular benchmark is good or bad is not easy: First, we need to separate the benchmark from its actual implementation, i.e., a poorly designed benchmark may be implemented perfectly and vice versa¹. Second, there is a number of design objectives, which we will discuss in the following sections – deciding whether a benchmark fulfills a particular design objective is not trivial. Third, these design objectives are often in conflict. For instance, there is a tradeoff between the affordability of running a benchmark and its complexity, expressiveness, or level of detail. However, rating the importance of objectives highly depends on the motivation for using the benchmark in the first place. For researchers, affordability will often not matter since they will be more interested in detailed results. In a business scenario, however, where such a benchmark is run repeatedly and frequently (e.g., as part of a build process), the price tag of running a particular benchmark suddenly becomes much more relevant.

What should also be noted is that design objectives and implementation objectives are often confused and mixed-up, both in literature and practice. This seems

¹ We will discuss implementation objectives in detail in chapter 8

to be due to the fact that both, practitioners and researchers, often do not distinguish the abstract notion of a benchmark and one or more instances of implementations: SPEC benchmarks are typically distributed as ready-to-run programs, which encourages this way of thinking. In contrast, however, TPC benchmarks typically come without an implementation, which is left to the person or entity running the actual benchmark.

In essence, *benchmark design* leads to a comprehensive specification whereas *benchmark implementation* will – building on such a specification – lead to an executable program. Objectives of both phases are without a doubt strongly interconnected; some objectives may even be objectives of both design and implementation phase. However, there are also objectives that fall into only one of the two categories. We, therefore, believe that it is important to differentiate both kinds of objectives – this chapter focuses on design objectives, we will cover implementation objectives in chapter 8.

In this chapter, we will start by discussing objectives of benchmark design – both general ones and more recent cloud-specific ones – before describing ideas on how to resolve conflicts between design objectives.

5.2 Objectives of Benchmark Design

A benchmark should fulfill a number of design objectives – of course, their individual importance varies depending on the motivation for running the benchmark. There are a number of publications on design objectives of benchmarks, e.g., [24, 29, 33, 39]. Most of them focus more on traditional systems benchmarking as, e.g., done by TPC or SPEC, and do not consider the specifics of cloud service benchmarking. Still, many of these design objectives are general enough to be applicable in the new context as well.

5.2.1 General Design Objectives

In this section, we describe well-established design objectives from traditional systems benchmark that are still applicable to cloud service benchmarking. The ideas in this section are based on [24, 29, 33, 39].

The first design objective is **relevance**. Relevance prescribes that a benchmark should be designed based on a realistic scenario and that the benchmark should interact with the SUT in a realistic and typical way. For instance, when benchmarking a database service, relevance implies that the benchmark should try not to emphasize query types that will only rarely occur in practice. Instead, the frequency of running particular queries should reflect frequencies that would be observable in actual applications from the scenario domain. A relevant benchmark will then stress features of the SUT in a way that is similar to real world applications. Other aspects

to consider in the scope of relevance are longevity, i.e., whether the application scenario is one that will continue to exist for some time to come, or target audiences, i.e., whether there is a sufficiently large group of people that will be interested in the results of such a benchmark.

The second design objective is **reproducibility** or **repeatability**. Reproducibility simply means that there is a certain confidence that running the exact same benchmark multiple times will yield the same or very similar results. Repeatability, which is a requirement for reproducibility, means that a benchmark can exactly reproduce what the SUT is confronted with. Especially in cloud contexts, where reproducibility may not always be possible, repeatability becomes an important design objective. In the past, both have been used interchangeably. Generally, this objective implies that more determinism in the benchmark specification is good for repeatability and, thus, for reproducibility – however, repeatability also bears a certain conflict potential since full determinism incentivizes system vendors or service providers to implement special “test situation only” configurations of their systems and services, i.e., the SUT will behave differently when it is benchmarked. Obviously, reaching both reproducibility and repeatability also highly depends on the benchmark implementation. However, there is also the aspect that two different implementations of a benchmark specification *can* actually produce comparable results and are guaranteed to do so in the absence of bugs or violated implementation objectives. This observation implies that benchmark specifications should not only focus on deterministic outcomes, but should also clearly specify all relevant aspects, i.e., they should be concise (so that they can still be understood) but also comprehensive, not fostering implicit assumptions.

The third design objective is **fairness**. Fairness means that a benchmark should treat all potential SUT fairly – it should not make implicit assumptions or overemphasize features that exist only in one solution. A benchmark must always be domain-specific, and while relevance is strengthened if a benchmark can be used with as many different cloud services as possible, this capability will always also affect fairness. For instance, it is not fair to use a benchmark which was designed for measuring performance of complex queries in RDBMS to demonstrate poor performance of a distributed file system. A fair assessment also means that – in contrast to traditional benchmarks – a benchmark should always have a multi-quality assessment: as we have seen in chapter 3, there are always tradeoff relationships between qualities. Comparing a service that is specifically optimized for a single quality to a general purpose cloud service will never be fair, unless all sides of the tradeoff are measured and assessed as part of the benchmark. An exception to this can be made when services are compared that have been optimized for the same purpose. However, even then this claim should be verified experimentally to assert fairness. Existing benchmarks developed by system vendors or service providers should always be taken with a grain of salt since these actors have incentives to design the benchmark in a way that favors their own product. In comparison, benchmarks designed by a larger group, including for example multiple major industry players, or benchmarks developed by researchers tend to be fairer – it all comes down to the

question whether the benchmark designers have intensive prior knowledge of only a subset of the available solutions or not.

The fourth design objective is **portability**. Portability means that the benchmark should be designed in a way that it can be run against a large number of systems and services. For instance, portability implies that the benchmark should, regarding the use of features, be “leading edge but not bleeding edge” [33]. This means that the benchmark designers must find a good tradeoff between using only a very small subset of features, thus, rendering the benchmark obsolete by design, and using cutting edge features, which may only be offered by a limited number of cloud services, which in turn will endanger portability. Portability is also affected by the economics of running a benchmark: benchmarks that require extensive implementation efforts with complex logic or that need to be deployed on expensive infrastructure (i.e., an infrastructure that may not be affordable for service consumers or even only some competitors) are not very portable. Historically, portability was especially important for database benchmarks that needed specific hardware infrastructure. Today, in the context of cloud services, portability implies that the benchmark design itself (and later its implementation) should not rely on the availability of advanced services offered by the same cloud provider. All in all, it should not only be *possible* to run implementations of the benchmark on multiple platforms against a variety of cloud services, it should also be *affordable* to do so.

Finally, the fifth design objective is **understandability**. Foremost, understandability means that the first impression of a benchmark should not be of the “what is this?” kind – interested parties should be easily able to understand precisely what the benchmark is measuring and should intuitively judge benchmark results to be verifiable. A key aspect of meeting this objective is having meaningful and understandable metrics (we will discuss this in detail in chapter 6) that are intuitive to understand and expressive enough to describe the analyzed behavior. However, understandability also means that the benchmark workload should follow an understandable application scenario – while there are cases where micro-benchmarks (see chapter 7) are necessary and helpful, their expressiveness and understandability is often impaired. Obviously, there is also a tradeoff between simplicity of a benchmark, which is good for understandability, and using complex features in a highly realistic way, which is good for relevance – the more realistic a benchmark represents an application scenario, the more difficult is it to understand but also to implement and to run it on different platforms (portability). There has been a trend towards more complex and comprehensive benchmark designs, which should be monitored carefully: For instance, 1989’s TPC-A benchmark has a 51 page specification whereas 2010’s TPC-C benchmark has a 132 page specification, or 2003’s TPC-W benchmark describes 8 entities with a total of 63 attributes whereas 2010’s TPC-C benchmark, which essentially describes the same application scenario, has 9 entities with a total of 92 attributes. Since all attributes and entities will be queried this is a good indicator of complexity.

5.2.2 New Design Objectives in Cloud Service Benchmarking

There are a number of new design objectives or challenges that need to be addressed in cloud service benchmarking. Some of these have already been discussed in literature [9, 14, 24], others have not been covered yet. We will try to give a comprehensive overview in this section. But first, we will discuss some of the characteristics of cloud services that motivate these objectives.

When trying to differentiate cloud services from traditional systems, there is a number of key differences, which may affect benchmarking: Typically, cloud services are no longer “static”. While traditional systems often tried to assess performance based on a fixed amount of resources, clouds are (theoretically) designed to be scalable and to provide the illusion of infinite resources. Often, cloud services are even designed to automatically adjust the amount of provisioned resources and to adapt to changing load patterns. Furthermore, performance metrics are frequently expressed in relation to the acquisition cost of the underlying hardware. However, cloud services with their pay-as-you-go on-demand pricing model have disruptively changed IT expenses from fixed cost to variable cost.

Another aspect is geo-distribution: cloud services are inherently geo-distributed, and their users likewise. This means that issues that were traditionally only considered by people working on distributed systems have now become mainstream. Even small cloud-based applications are confronted with challenges like reduced consistency guarantees from underlying storage services, VMs that are not guaranteed to be in the same rack, or VMs that lack dedicated network interconnection. At the same time, geo-distribution offers new opportunities regarding fault-tolerance of applications or simply reduced latencies for end users by having application endpoints in various geographic regions. Still, quality in the cloud is subject to much larger fluctuations of which the cloud provider may not even be aware of [11] – especially so for more complex qualities beyond performance.

Multi-tenancy is another fundamental aspect of cloud services: users of cloud services operate on shared resources that offer varying degrees of resource isolation. This situation makes it much harder to reproduce cloud-based experiments. Furthermore, measured values are bound to change over time since not only is the total amount of resources that a cloud service uses not constant, but also the mapping of resources to cloud service consumers will change frequently.

There is also a plethora of cloud services that essentially offer the same basic functionality, but highly deviate regarding their set of additionally offered features. Since cloud services are available globally to all paying users, i.e., the potential customer base is gigantic, it generally pays off for providers to also cater for the needs of the long tail of business. While general purpose solutions are still available, specialized solutions are far more widely available. For instance, RDBMS used to be the de-facto standard for database systems – today, however, NoSQL systems and comparable cloud services are offered in a variety of flavors ranging from column stores and key-value stores to graph databases or document stores. And even within these categories, systems and services are much more diverse than they used to be. For example, Cassandra [46] differs way more from Amazon DynamoDB than

MySQL ever did from PostgreSQL. The same is true for compute services which come in a variety of flavors. All this, however, strongly affects relevance and portability: it is much more difficult to identify the group of cloud services targeted by a particular benchmark than it used to be.

Finally, the nature of modern applications needs to be considered: Today's applications are evolving at an incredible speed, supported by agile development methodologies. Obviously, this development speed has a strong impact on how applications interact with underlying cloud services – every change to the application may also disruptively change the requirements on underlying infrastructure but also the kind of stress that an application creates on a cloud service. In consequence, benchmarks that have been an exact representation of an application yesterday need not necessarily be relevant tomorrow. Therefore, benchmarks must be able to adapt to changing environments just as quickly.

Building on these considerations, we recommend the following strategies when designing cloud service benchmarks:

1. Benchmarks should make as little assumptions as possible and should also try to avoid strict quality requirements. Instead, it is much better to identify an ideal case and measure deviations from that case. This approach implies that an ideal cloud service benchmark will always be a multi-quality benchmark.
2. Novel, complex qualities require new measurement approaches and metrics to be benchmarked. To a certain degree, there needs to be a co-design of benchmark, measurement approach, and metric to assert meaningfulness of results.
3. Benchmarks should account for failure situations, i.e., they should test how the cloud service fares when confronted with failures. However, this may not always be possible as cloud service benchmarks are rarely white box benchmarks – typically, they are black box or gray box benchmarks in a very dark shade of gray.
4. Cost metrics need to be defined per request or user interaction for all benchmarks that mimic transaction processing applications or per batch unit in case of analytic batch processing applications. Different complexity of operations may lead to different cost per request or per batch unit so that cost will typically be expressed as a vector. Benchmarks can also try to identify granularity of scaling steps, i.e., the discrete amount of resources that is added or removed while scaling: Larger scaling steps can be leveraged economically.
5. Since modern application deployments are typically geo-distributed, benchmark design should consider the distribution of both measurement clients and subunits of the SUT to be an integral part of the application scenario definition.
6. Benchmarks should not only measure new qualities, but also try to specifically provoke stress situations for these qualities. For instance, variable load

patterns are a good way for measuring scalability and elasticity of cloud services.

7. Benchmarks should be designed to capture data as detailed as possible. This means carefully choosing all configuration choices made but also to consider (mis)using standard monitoring tools to keep track of resource consumption and cost across machines, services, and software components. For instance, in the case of storage service benchmarking, it is highly critical to avoid having the performance bottleneck within the measurement clients, which would effectively measure performance of the client machine(s). Standard monitoring tools can help to identify problematic situations. Benchmarks should also never only record aggregate values.
8. Another, closely related aspect is to avoid designing benchmarks that measure something else than what is intended. For instance, in [70], the authors claim to study the feasibility and performance impacts of implementing client-side encryption for cloud database systems. However, the way they designed their experiments they primarily measure the compute performance of the client machine.
9. Today, applications are undergoing fundamental changes at an unprecedented speed. This means that existing benchmarks will quickly become obsolete. One way to address this challenge is to provide configuration parameters and tuning knobs that allow to alter benchmarks over time as well as to perform goal-oriented experimentation through micro-benchmarks (e.g., in YCSB [19]).
10. Since cloud services often do not only adapt provisioned resource amounts over time, but are even expected to do so, benchmarks should always be long-running experiments and should be repeated at different times of day and different days during the week to detect (a) stabilized behavior in long-running applications, (b) short term effects and their durations, and (c) seasonal patterns. We discuss the analysis of such patterns in section 12.4.1.

5.3 Resolving Conflicts between Design Objectives

Resolving conflicts between benchmark design objectives in a general way is hard if not impossible; too much depends on the motivation for benchmarking, the application scenario in mind, and the cloud service(s) in question. Furthermore, there are also transitive relationships and cross-effects between different design goals – in fact, comparable to quality tradeoffs, design objectives have interdependencies. Figure 5.1 gives an overview of some of the conflicts described in this chapter – additional ones exist and often also depend on the concrete use case. See the following examples which describes how we have solved these conflicts in the past.

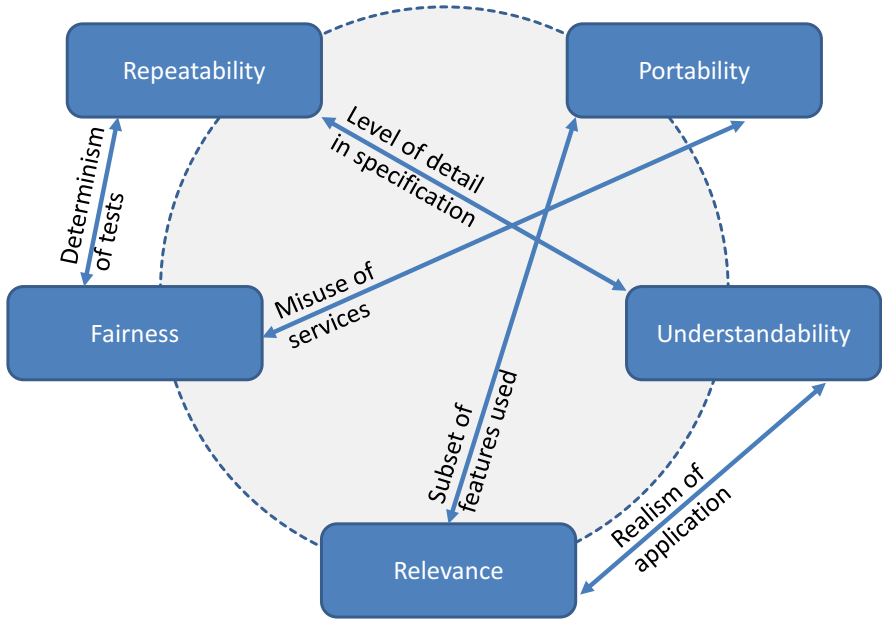


Fig. 5.1: Examples of Conflicts between Benchmark Design Objectives

Always consider all design objectives, never ignore any of them. Then, make a conscious decision to trade less important aspects for more important design goals.

Benchmarking Performance Impacts of Security Settings: When we benchmarked the performance impacts of enabling data-in-transit encryption for NoSQL stores [53,55,56], we aimed to better understand when and how encryption, e.g., through TLS, affected the throughput that the SUT could handle. Since we were not interested in a particular application workload but rather only needed to fully load the system both with and without security settings enabled, we opted for YCSB-based workloads [19] to stress the target system. This way, we traded “relevance” aspects that were less important to our initial question (realistic application workload) for “portability” (benchmarking system support for a variety of storage services) and “understandability” aspects (very simple workload).

Consistency Benchmarking: In our work on consistency benchmarking [10, 11, 13], we wanted to experimentally determine probabilistic upper bounds for staleness in eventually consistent [7, 69] data stores. Normally, “relevance” demands that the benchmark should be designed in a way to closely resemble a realistic application. However, in this case, we were not interested in the behavior that a specific application might observe. Instead, we wanted to measure the behavior that the (from the perspective of the data store) theoretically worst possible application would experience. Therefore, we decided to ignore the realism aspects of relevance (of course, upper bounds are still of interest to a large potential target group) and designed the benchmark workload in a way that it would stress the storage service in a way that all kinds of negative behavior would be observable. In this example, we thus solved a tradeoff between different aspects of relevance.

Additionally, it could have been interesting to identify inconsistency phenomena, e.g., across database tables. Still, we decided to use a workload that would require only a key-value interface with a read and an update operation. This may not be 100% fair towards storage services which were then used in a way that they were not specifically designed for (but neither specifically *not* designed for), however, it helped towards portability and understandability.

Cloud Service Benchmarking

Measuring Quality of Cloud Services from a Client
Perspective

Bermbach, D.; Wittern, E.; Tai, S.

2017, XIV, 167 p. 25 illus., 11 illus. in color., Hardcover

ISBN: 978-3-319-55482-2