PolyLayer: the next 700 storage configurations

Workshop Paper

João Lopes
InvisibleLab
joao.lopes@invisiblelab.dev

Bruno Pereira
INESC TEC & U. Minho
bruno.f.pereira@inesctec.pt

Filipe Pereira
INESC TEC & U. Minho
filipe.s.pereira@inesctec.pt

Vicente Muñoz
InvisibleLab
vicente.munoz@invisiblelab.dev

Tiago Gomes
InvisibleLab
tiago.gomes@invisiblelab.dev

Rui Ribeiro
InvisibleLab
rui.ribeiro@invisiblelab.dev

Filipe Costa
InvisibleLab
filipe.costa@invisiblelab.dev

Marta Bonjardim
InvisibleLab
marta.bonjardim@invisiblelab.dev

Francisco Cruz
InvisibleLab
francisco.cruz@invisiblelab.dev

João Paulo
INESC TEC & U. Minho
joao.t.paulo@inesctec.pt

Francisco Maia
Universidade do Porto & INESC TEC
franciscomaia@fe.up.pt

2025 44th International Symposium on Reliable Distributed Systems (SRDS) | 979-8-3315-9199-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/SRDS69199.2025.00052

***Abstract*—Modern storage systems requirements demand flexible, scalable solutions that address diverse concerns such as data reduction, replication, security, and multi-cloud distribution. Existing solutions often provide these guarantees through monolithic implementations, limiting their adaptability to specific application needs.**

This paper introduces PolyLayer, a multi-interface, composable and multi-backend storage architecture. It builds on the concept of stackable storage architectures and redesigns these to support commonly used user APIs (*e.g.*, POSIX, Key-value, Object store), while providing support for data persistence across multiple storage backends (*i.e.*, on-premises, cloud services, blockchain).

We present the first steps towards the design of such architecture, while implementing a proof-of-concept and evaluating it. Our preliminary results show that the design can effectively be used in real-world scenarios where new functionality is added to a storage system with low overhead over the base system. For instance, we show how anti-tampering mechanisms can be added to a traditional relational database without any change to the database itself or the application using it.

***Index Terms*—Data systems, Databases, Distributed databases.**

I. Introduction

The exponential growth of digital data and the increasing diversity of storage requirements have driven the development of many data storage solutions [1–7]. Modern applications—ranging from cloud-native services to edge computing platforms—demand storage systems that are not only scalable and reliable but also adaptable to specific operational and regulatory needs (e.g. HIPAA, GDPR). Traditional monolithic storage architectures often struggle to meet these evolving demands, as they are typically designed with fixed feature sets (e.g. richness of API, fault-tolerance capabilities, security configurations, etc.) and limited flexibility. For instance, adding anti-tampering guarantees to a traditional relational database without impactful changes to the application is non-trivial.

In response to these challenges, we introduce PolyLayer, a novel approach to data storage that combines three desirable characteristics of storage systems: *i)* client API flexibility providing compatibility with POSIX (legacy interface), key-value (used by several services and applications, for instance at Meta), and object store (the default API of cloud storage services provided by Google, Amazon, and Microsoft) interfaces; *ii)* a stackable and configurable design; and *iii)* multi-backend support including cloud storage services, blockchain, and on-premises storage. Inspired by previous approaches, PolyLayer decomposes the storage stack into discrete, reusable layers, each addressing a specific concern, such as data reduction, replication, security, or multi-cloud distribution. Unlike any existing solution, the layer system is completely independent from the actual interface exposed to client applications. Following this design, PolyLayer is able to be coupled with virtually any application or serve as middleware for a variety of storage systems. Additionally, by allowing these layers to be flexibly combined, PolyLayer empowers system architects to construct bespoke storage solutions optimised for their unique use cases. The stackable design is then enhanced with the capability of configuring different data persistence layers, which allows PolyLayer to store data across cloud storage services, local services, or blockchain infrastructures.

Independently, the three aforementioned characteristics are not new and have been addressed by systems such as SafeFS [8] or OpenDal [6]. However, to the best of our knowledge, this is the first system to combine all three in a single and FUSE-independent design.

This paper presents the design principles, architecture, and preliminary implementation of PolyLayer. To illustrate PolyLayer's capabilities, we demonstrate how its modular layering approach facilitates the creation of tailored storage solutions by offering preliminary results of its effectiveness in a real-world use case. Using FUSE (Filesystem in Userspace) [9] as

2575-8462/25/$31.00 ©2025 IEEE
DOI 10.1109/SRDS69199.2025.00052

one instance of a PolyLayer interface, we show how PolyLayer can be used to seamlessly combine the capabilities of two different storage systems (Relational Database Management Systems and Blockchain systems) into a single system with anti-tampering capabilities.

II. Related Work

An early attempt to reuse distinct storage optimizations (*e.g.*, data compression, encryption, caching) implemented within different file systems has been proposed through a stackable file system design [10]. Such a design includes multiple layers sharing a standard interface and implementing specific storage functionalities. A similar design, but for the block device interface, can be found on the device mapper framework [11].

A significant drawback of the aforementioned solutions is that layer functionality is implemented within kernel space, making this a complex, error-prone, and time-consuming task [12]. This motivated the proposal of the SafeFS user-space stackable file systems, where storage optimizations are fully developed in user-space by leveraging the FUSE framework [8]. While the latter framework enables quicker and more reliable implementation of layer functionality, it is tightly coupled to FUSE and its inherent performance drawbacks [13]. Recently, stackable designs have been extended to other user-space frameworks, for instance, to Intel SPDK, a kernel-bypass framework for accessing NNMe SSD devices which allows implementing and combining block-oriented storage optimizations as layers [14].

In parallel, the number of storage interfaces (*i.e.*, block device, file system, key-value store, object store) and services (*e.g.*, Cloud storage, Blockchain storage) has also been increasing. Unifying these different storage interfaces and porting applications to transparently use them is therefore another important challenge towards unifying storage systems, which is not addressed by the stackable designs discussed above.

In this field, OpenDAL has emerged as an interesting open-source framework that stands out by delivering a unified API for multiple storage system interfaces through custom connectors rather than vendor-provided SDKs, ensuring consistent behaviour across different storage types without sacrificing performance [6]. Interestingly, OpenDAL is unable to leverage the stackable design and layer reuse of previous work when dealing with multiple storage connectors (*e.g.*, using the same compression or data encryption layer when replicating data across multiple cloud and/or blockchain services). Further, the lack of a stackable and shared layer architecture leads to synchronisation and consistency challenges across connectors that maintain separate configurations and processing logic.

Polylayer differs from the previous work as it aims to leverage and combine strong key aspects of the previous work, namely: stackable layered design, user-space driven layer implementation, unified storage interface support, while being independent from a given framework (*e.g*, FUSE) or even API.

III. PolyLayer Architecture

PolyLayer is inspired by the design of SafeFS [8], which provides a modular, layered architecture for building a File System. However, modern data systems often require a combination of different storage systems, such as databases, object stores, and file systems, to meet diverse requirements. Such a situation leads to very complex architectures, which are challenging to maintain and evolve. In addition, even if each data storage system has different desirable features, combining them to provide additional functionalities to applications is not straightforward. Figure 1 depicts the architecture for PolyLayer.

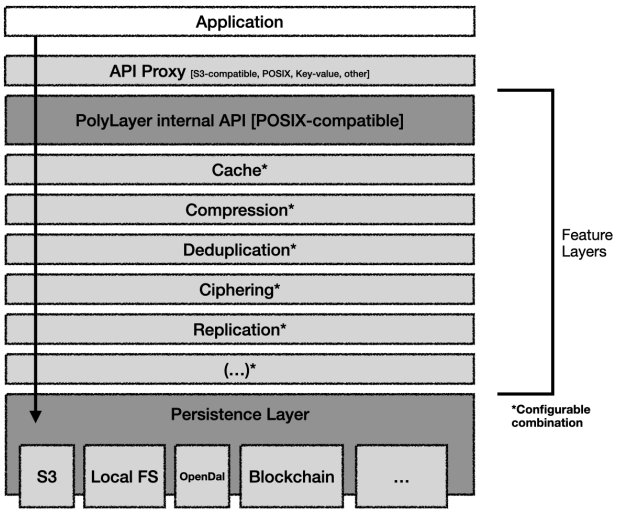

Fig. 1: PolyLayer high-level architecture.

The design of PolyLayer follows three main principles:

- **Flexible application interface** - the system must be compatible with a wide array of applications and systems.
- **Configurability** - adding or removing functionality must be straightforward and non-disruptive to the applications using PolyLayer.
- **Agnostic of persistence medium** - PolyLayer must allow the use of state-of-the-art data storage systems or libraries without changes to the core functionality or the applications using it. It should be possible to change from, for example, an S3 bucket [15] to a Google File System [16] or accommodate API improvements offered by upgrading a local file system.

We address these principles in the following way. PolyLayer was designed to export, unlike previous approaches, a configurable API depending on the client application. PolyLayer supports S3-compatible interfaces, Key-value interfaces, and custom interfaces. This configurable API is then translated into an internal, unified API, implemented by each layer. Since layers implement the same API, they are interchangeable and composable into a stack of layers, and the combination of

layers into a specific stack yields a set of capabilities offered by PolyLayer.

The layer stack is built using two types of layers: *Feature Layers* and *Persistence Layers*. Feature Layers are called to perform operations on data and then delegate the call to the next layer in the stack. For example, a ciphering layer would get plaintext data and output ciphered data to the next layer in the stack. Such behavior transparently adds a security layer to the storage system and is independent of other concerns, such as data replication or data persistency. In fact, layers should be designed to be agnostic of the specifics of other layers (e.g. configuration, performance, dependencies, etc.). Persistence Layer can be called at any time by any of the Feature Layers, and at least one Persistence Layer is required at the bottom of the stack. These layers offer a means of persistently storing data in some storage medium (e.g., local disks) or service (e.g., cloud storage service). Unlike previous approaches, there is not requirement on the type of system or service for the implementation of a persistence layer. A persistence layer may be implemented by a cloud storage service such as Amazon Dynamo [3], Amazon S3 [15], a blockchain system such as Solana [17], or even a distributed file system deployed on premises.

This design is extremely flexible, and different combinations of layers will yield different capabilities for the same exposed API. Note that PolyLayer's design allows it to function as an effective middleware between applications and their data persistence mediums. Moreover, complex capability additions to applications become straightforward. For instance, an application that uses S3 for its data backend can easily add geographical distribution by using a *replication* layer in combination with two geographically dispersed persistence layers. Similarly, an application using a costly cloud service can easily migrate to a different one, even if their API's are significantly different, without having to change any of its code by installing PolyLayer configured with the legacy API as the top layer and the new data backend as the persistence layer.

IV. PolyLayer Prototype

In order to demonstrate the capabilities of PolyLayer, we have implemented a system prototype and have evaluated it using a real-world use case (Figure 2).

PolyLayer's prototype relies on the implementation of four core operations at each layer: `open`, `close`, `pread`, and `pwrite`. Layers include an additional initialization operation that receives layer-specific arguments provided via a TOML configuration file. Each layer is configured with an identification key, a type (selected from the implemented layer types), and type-specific arguments, which may include credentials and a connection to the next layer to call in the stack. The user specifies, through configuration, which will be the *root layer* and the order in which layers will be stacked. In the concrete instantiation of Figure 2, the external interface is offered by FUSE, the root layer is an anti-tampering one, and there is a demultiplexer layer using two persistence layers. One is an Amazon S3 bucket service, and the other is a local file system. This specific configuration is implemented as follows.

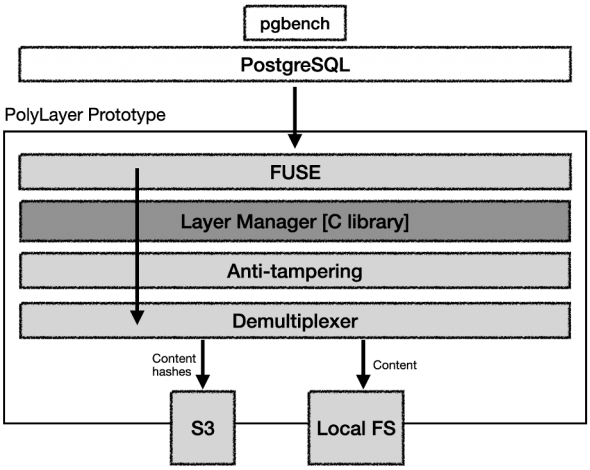

Fig. 2: High-level overview of PolyLayer prototype.

A. Demultiplexer Layer

The Demultiplexer Layer enables the stack to fork into N parallel paths, allowing the user to split the stack into N branches, each with different layer configurations and potentially different Persistence Layers, enabling multi-backend storage. In this case, we will be using a local file system and a remote cloud service.

This layer does not enforce data consistency between the N parallel paths, which should be the concern of a specific layer. In this scenario, the demultiplexer layer is solely concerned in offering a way to store data in different backends, which will allow the anti-tampering layer to make use of both an S3 bucket and a local filesystem. Operations on the N parallel paths are made concurrently.

B. Anti-Tampering Layer

Often, systems operate in multi-tenant environments or multi-application environments. In most applications, it is important to understand if the data being read is, in fact, valid data produced by the same application some time before, and not the result of unwanted access and tampering of data. The idea behind the anti-tampering layer implemented in PolyLayer is to add an anti-tampering detection mechanism to any storage system transparently. This is done as follows. The Anti-Tampering Layer hashes all content written to the storage system and stores the resulting hashes in a separate immutable data store—potentially a blockchain—to leverage its inherent anti-tampering capabilities. Any time the application tries to read data, a comparison between the stored hash and the hash of the content being read is made. If a discrepancy is detected, then a warning is issued indicating potential data tampering. This hybrid approach allows us to transparently

combine the anti-tampering guarantees of a blockchain with the performance of more traditional systems.

C. Persistence Layer

The Persistence Layer is implemented as a Rust library that provides a similar API over diverse storage services, enabling a unified interface across backends with differing specifications. [1] This layer exposes access to a diversity of storage backends to all the other layers in the system.

PolyLayer is compatible with the OpenDAL library to handle low-level calls to storage services, which significantly reduces the engineering effort required to add new storage systems, but it is not restricted by it. In fact, PolyLayer even supports multiple implementations for the same backend to accommodate differences in performance characteristics for specific operations. For example, PolyLayer supports both `s3_opendal`, which is the OpenDal implementation of an S3 SDK, and the official `s3_aws_sdk`.

In our prototype, the persistence layer offers access to three backend services. The `s3_opendal` offers access to the S3 OpenDal implementation, `solana_sdk` offers access to Solana's blockchain infrastructure, and `local_fs` allows access to local file systems.

Naturally, not all storage systems support the same operations or are suitable for every use case. Persistence layer components need to be adapted for each storage service. For example, in the `solana_sdk`, the `pwrite` operation is implemented with limitations: data is stored in the transaction memo, limited to 180 bytes per transaction, which is a Solana technical limitation. However, we can still leverage this service for hash and metadata storage rather than full file storage, supporting the anti-tampering layer.

V. Use cases

To illustrate the applicability of PolyLayer we describe two real-world use cases. We then use one of them for our preliminary evaluation described in Section VI.

a) RDBMS optimisation: Often, web applications rely on a single RDBMS for their data storage needs. For convenience, system administrators deploy the RDBMS in the Cloud, which results in compute and storage costs.

Let us take, for instance, the case of using Amazon AWS. To deploy a RDBMS, one typically asks for an EC2 instance and, optionally, some external storage or deploys a managed service such as AWS RDS. Albeit being the more immediate approach, using managed RDBMS such as AWS RDS is not the preferred approach for many small and medium sized companies since using managed services might lead to variable monthly costs. Having predictability and controlled costs is a must-have for these companies that prefer to allocate EC2 instances with a fixed monthly fee. In this scenario of using an EC2 instance, the instance is typically coupled with the Elastic Block Storage service, and each type of EC2 instance will allow up to a certain amount of EBS storage. For the typical web application workload, EBS storage is effective and avoids extra costs of the expensive AWS RDS or other types of dedicated storage systems. In fact, even avoiding RDS costs, configuring external storage as the persistence medium for an RDBMS running in EC2 can still lead to increased costs and, more concerning, can also lead to performance drops. Performance drops come from the extra network layer between the RDBMS and the storage medium (EBS is typically local to the EC2 instance), and adding better network connectivity and/or faster storage mediums will, again, demand additional costs. Naturally, companies tend to avoid costly configurations and, whenever possible, rely on the EC2 + EBS combination to use resources efficiently.

Interestingly, web applications are also very data-intensive applications. Running a web application is often not a very CPU-intensive task, but data storage requirements are ever-increasing. This leads to a difficult resource administration challenge. The increase in storage space required to run the application will, inevitably, require the allocation of extra storage. If done at the EC2+EBS instance, it will mean paying for superfluous and expensive computational power.

Enter PolyLayer. Deploying a data deduplication or data compression layer using PolyLayer between the RDBMS and the EBS storage will help reduce storage requirements, maintain low costs, and avoid any changes to the application.

b) Anti-tampering: Ensuring data security in cloud computing or multi-tenant environments is increasingly challenging. Solutions such as blockchain data stores, which have anti-tampering security out of the box, are still very limited in terms of performance and capability to store large datasets. In order to add anti-tampering capabilities to an existing system, significant changes to the applications or the infrastructure deployment in use are required.

Enter PolyLayer. With PolyLayer, it becomes straightforward to add an anti-tampering layer to an existing data storage system without having to change any application or interfere with the existing infrastructure. Moreover, PolyLayer allows the use of blockchain capabilities without having to incur in privacy risks or significant performance penalties. In the next section, we demonstrate how PolyLayer can be configured and run for such a use case.

VI. Preliminary results

For a preliminary validation of PolyLayer's prototype we focused on the novel Anti-Tampering capabilities as depicted in Figure 2. To test such a configuration, we chose to run a popular relational database management system on top of PolyLayer (PostgreSQL) to address a common application use case. As the interface layer, we integrated PolyLayer with FUSE, which allowed us to run PostgreSQL directly on PolyLayer, without requiring any modifications to PostgreSQL itself.

The question we wanted to answer was whether we could run such a setup without seeing problems with PostgreSQL and assess the overhead caused by the anti-tampering system. To achieve this, we used `pgbench` [18] to benchmark the

[1] Layers can be implemented in virtually any programming language, provided that appropriate C bindings are provided.

PostgreSQL database initialized in a directory mounted via FUSE, using three PolyLayer configurations: *i)* a local layer only (baseline), *ii)* the Anti-Tampering layer with hashes stored locally (to evaluate the anti-tampering overhead), and *iii)* the Anti-Tampering layer with hashes stored in an S3 bucket (to assess the asynchronous nature of the anti-tampering system).

For the tests, we used an AWS EC2 instance of type `m7i.xlarge`, equipped with 4 vCPUs and 16 GB of RAM. For configurations involving S3 storage, we configured a VPC Gateway Endpoint to reduce access latency.

Each test was performed with a scale factor of 50 and consisted of five consecutive runs of 30 minutes. We evaluated three workload types: a mixed read/write workload, read-only (`SELECT` operations), and write-only (`INSERT` operations). To isolate the overhead introduced by FUSE, we also executed the same workloads on PostgreSQL using the local file system directly, without FUSE.

The results are summarized in Table I (throughput) and Table II (latency), both showing the averages of the runs. Looking at these preliminary results, we can observe the following.

A. Local vs. FUSE+Local

Under the mixed read/write workload, the FUSE configuration exhibited an 11% reduction in average transactions per second. For the read-only workload, the performance difference was more pronounced, with a 60% performance drop. Under the write-only workload, we observed a 17% decrease in the number of transactions executed each second. These numbers show us that FUSE introduces a baseline overhead, which was expected and now allows us to analyze the overhead of PolyLayer.

B. FUSE+Local vs. FUSE+Anti-Tampering (Local Hash)

To evaluate the overhead introduced by adding additional security guarantees, we ran tests using the Anti-Tampering layer with hashes stored locally. For the read-only and write-only workloads, both configurations exhibited comparable transactions per second and latency, which shows that no significant overhead is being added when we can cache the comparison information. A different scenario is observed for the mixed read/write workload, where the system has a performance drop of roughly 33%. This is due to the fact that writes are now forcing the computation of new hashes and new anti-tampering checks. Such an overhead could be prohibitive for demanding applications but we believe that there is plenty of margin to improve the performance of the system under these circumstances. This early-stage implementation of PolyLayer is file-oriented, which hinders its performance for workloads that update data. A block-based implementation of the storage layers is one of the improvements that we expect to have a significant impact in the performance of the system. Another direction of improvement is the use of alternative client APIs and avoiding FUSE, which in itself adds significant overhead to the system.

C. FUSE+Anti-Tampering (Local Hash) vs. FUSE+Anti-Tampering (S3 Hash)

To evaluate the overhead introduced by using an external, over the network, backend, we ran tests using the Anti-Tampering layer with hashes stored in an S3 bucket. As expected, the introduction of a network hop has introduced additional performance penalty. The S3-backed configuration showed an additional 18% reduction in transactions per second in the read/write mix workload.

These preliminary results are interesting in essentially two ways. First, they reinforce our belief that the approach taken for PolyLayer is promising. The flexibility of the design allows for a very wide range of configurations, which can address real-world scenarios such as the need for anti-tampering guarantees in multi-tenancy environments. Second, even if in our tests we have observed non-negligible overhead, the fact that such overhead is manageable for read-only workloads provides us the insight that this approach might be feasible for content delivery applications (e.g. CDN, social media platforms). Additionally, for production-level configurations, there is room for improvements and the use of alternative technology such as LD_PRELOAD for PostgreSQL, similarly to what was done for other applications [19], is a promising research path.

VII. Discussion

PolyLayer demonstrates a new direction for data storage system design by enabling modular, stackable, and highly configurable storage architectures. The system's layered approach addresses the growing need for flexibility in modern data environments, where requirements for security, replication, or multi-cloud support can vary significantly across applications and evolve over time.

It is, to the best of our knowledge, the first system of its kind to separate a configurable API from the stackable system and the first to be compatible with heterogeneous data persistency systems. PolyLayer also introduces the hybrid anti-tampering mechanism, which enables guarantees only seen in blockchain systems applied in combination with traditional RDBMS. This materializes a new class of storage systems, which are particularly useful for multi-tenant environments.

Our preliminary evaluation highlights several important observations. First, PolyLayer's configurability and modularity allow system architects to compose storage stacks tailored to specific needs, combining feature layers such as anti-tampering and demultiplexing with diverse persistence backends. This modularity simplifies the integration of new capabilities and supports rapid adaptation to changing requirements. In terms of performance, the measured overheads for our preliminary experiments remain within acceptable bounds, given the proof-of-concept nature of our FUSE-based prototype, while there is margin for optimization.

PolyLayer provides a foundation for the next generation of data storage systems, where configurability and composability are first-class citizens. By decoupling storage concerns into reusable, user-space layers and supporting a unified

	Read/Write Mix	Read-Only	Write-Only
No Fuse	3602.04 ± 14.90	85943.15 ± 291.07	5020.75 ± 3.62
Local	3215.36 ± 31.33	34709.66 ± 256.66	4141.10 ± 55.51
Anti-Tampering Local Hash	2159.81 ± 30.52	35281.88 ± 237.89	4444.91 ± 357.23
Anti-Tampering S3 Hash	1790.32 ± 15.17	33607.99 ± 1506.85	3827.04 ± 33.37

TABLE I: *pgbench* throughput (transactions per second) results for the configured workloads and PolyLayer configurations.

	Read/Write Mix	Read-Only	Write-Only
No Fuse	2.767 ± 0.012	0.113 ± 0.001	1.988 ± 0.002
Local	3.10 ± 0.03	0.287 ± 0.002	2.40 ± 0.03
Anti-Tampering Local Hash	4.62 ± 0.07	0.283 ± 0.002	2.26 ± 0.17
Anti-Tampering S3 Hash	5.58 ± 0.05	0.296 ± 0.014	2.61 ± 0.02

TABLE II: *pgbench* measured average latency (milliseconds) for different workloads and PolyLayer configurations.

interface across heterogeneous backends, PolyLayer enables organisations to efficiently meet diverse and evolving storage requirements. The architecture's flexibility, combined with its demonstrated effectiveness in real-world scenarios, positions PolyLayer as a promising solution for modern data storage challenges.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback, which helped to improve the paper. This work is co-funded by the European Regional Development Fund (ERDF) through the NORTE 2030 Regional Programme under Portugal 2030, within the scope of the project BCD.S+M, reference 14436 (NORTE2030-FEDER-00584600).

REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[2] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[4] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[5] F. Maia, M. Matos, R. Vilaça, J. Pereira, R. Oliveira, and E. Rivière, "Dataflasks: Epidemic store for massive scale systems," in *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS '14)*. IEEE Computer Society, 2014, pp. 79–88.

[6] "OpenDAL," accessed: 2025-07-10. [Online]. Available: https://opendal.apache.org/

[7] D. Trautwein, A. Raman, G. Tyson, I. Castro, W. Scott, M. Schubotz, B. Gipp, and Y. Psaras, "Design and evaluation of ipfs: a storage layer for the decentralized web," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 739–752. [Online]. Available: https://doi.org/10.1145/3544216.3544232

[8] R. Pontes, D. Burihabwa, F. Maia, J. a. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira, "Safefs: a modular architecture for secure user-space file systems: one fuse to rule them all," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3078468.3078480

[9] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To fuse or not to fuse: performance of user-space file systems," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, ser. FAST'17. USA: USENIX Association, 2017, p. 59–72.

[10] J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 1, pp. 58–89, 1994.

[11] "Device mapper," accessed: 2025-07-10. [Online]. Available: https://en.wikipedia.org/wiki/Device_mapper

[12] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra incognita: On the practicality of {User-Space} file systems," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[13] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {FUSE} or not to {FUSE}: Performance of {User-Space} file systems," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 59–72.

[14] "The storage performance development kit (spdk)," https://spdk.io, accessed: 2025-06-16.

[15] Amazon Web Services, Inc., *Amazon Simple Storage Service User Guide*, 2024, retrieved July 13, 2025. [Online]. Available: https://docs.aws.amazon.com/s3/

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, 2003, pp. 29–43.

[17] A. Yakovenko, "Solana: A new architecture for a high performance blockchain," https://solana.com/solana-whitepaper.pdf, 2018, accessed: 2025-07-13.

[18] PostgreSQL Global Development Group, *pgbench: Run a benchmark test on PostgreSQL*, 2025, postgreSQL 17 Documentation. Retrieved July 13, 2025. [Online]. Available: https://www.postgresql.org/docs/current/pgbench.html

[19] R. Macedo, Y. Tanimura, J. Haga, V. Chidambaram, J. Pereira, and J. Paulo, "PAIO: General, portable I/O optimizations with minor application modifications," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*. Santa Clara, CA: USENIX Association, Feb. 2022, pp. 413–428. [Online]. Available: https://www.usenix.org/conference/fast22/presentation/macedo