A SUPPLEMENTARY FEATURES OF FLSTORE

Modular design FLStore's modular design enables seamless integration with existing FL frameworks without modifying clients or aggregators. Training can proceed unchanged, while client updates and metadata received by the aggregator are asynchronously relayed to FLStore's cache. FLStore then serves as a scalable and efficient storage solution, handling non-training tasks.

Multi-tenancy The serverless computing paradigm inherently provides isolation (Amazon Web Services, Inc., 2024a; Ellis & Contributors, 2024), allowing each user to create an isolated cache on the same FLStore instance. This enables customized caching policies per non-training workload/application, allowing FLStore to handle requests from multiple users simultaneously.

A.1 Scalability of FLStore

To demonstrate FLStore's scalability, we simulated increasing concurrent non-training requests, with FLStore maintaining 5 cached function instances (red line, Figure 11). We varied the number of concurrent client requests from 1 to 10 across six representative non-training workloads using the EfficientNet model. As shown in Figure 11, latency and cost remain nearly constant when concurrent requests are equal to or fewer than the cached functions. For 1 to 5 requests, the average latencies were 1.05 seconds for Malicious Filtering, 0.031 seconds for Cosine Similarities, 1.039 seconds for Scheduling (clustered), and 6.067 seconds for Clustering. Even with 6 and 7 requests, there was minimal increase in latency or cost. For 8 to 10 requests, latencies start increasing. However, this can be easily mitigated by scaling cached functions (creating copies of already cached functions) linearly with the number of requests, which incurs minimal additional cost, as discussed next.

A.2 Fault Tolerance

We evaluated FLStore's fault tolerance by testing ten different workloads using the EfficientNet model and sending 3000 requests over 50 hours. Faults (function reclamations) were generated based on the Zipfian distribution, observed in measurement studies on AWS Lambda (Wang et al., 2020). Figure 12 shows that with only 1 function instance, latency and cost are highest, with improvement as the number of replicas increases. With 3 to 5 function instances, latency and cost remain nearly constant, despite faults. In particular, 3 instances reduce latency by 50-150 seconds per request compared to a single instance in the face of faults.

Interestingly the cost of maintaining function replicas is negligible compared to the overhead and cost of re-computation and communication due to faults. For 50 hours and 3000 requests, maintaining 5 replicas costs just \$0.003, or \$0.000001 per non-training request served (Figure 13). In contrast, fewer instances lead to higher overhead and costs while maintaining more replicas reduces these costs by up to 3000×. Notably, we did not evaluate the impact of regular pinging, as this has already been explored in prior works (Zhang et al., 2023a; Wang et al., 2020).

B LATENCY AND COST PERFORMANCE BREAKUP

To identify the bottleneck, we broke up the accumulated latency between communication and computation time over 50 hours of experiments for the 10 different workloads.

B.1 FLStore vs ObjStore-Agg

Figure 14 shows the results with both communication and computation time for the ObjStore-Agg and only computation time for FLStore because communication time for FLStore is negligible in comparison due to co-located data and compute planes. The major bottleneck in ObjStore-Agg is Communication, in comparison the I/O time from memory to CPU is negligible (NinjaOne, 2024). For some workloads such as Inference, Debugging, and Scheduling, the difference between computation and communication times is significant. During inference communication consumes an average of 98.9% of time. This shows that current methodologies for computing non-training workloads for distributed learning techniques such as FL are significantly communication-bound. Thus, the reduction in communication times as brought by FLStore significantly improves the efficiency performance, which can be observed in Figure 14. We can also observe that FLStore provides significant improvements for smaller models, which is why FLStore is suitable in cross-device FL settings (Kairouz et al., 2019; Abdelmoniem et al., 2023). Across 50 hours and 3000 total requests we see Resnet18 with an average 82.04% (35.50 second) decrease in latency, MobileNet has an average 47.33% (75.99 second) decrease in latency, EfficientNet has an average 50.44% (100.18 second) decrease in latency, and Swin has an average 20.45% (4.42 second) decrease in latency. Thus, FLStore can significantly improve non-training tasks in FL with reduced latency. We next observe the reduction in total cost with FLStore.

We perform the same breakup analysis on the costs in Figure 15, showing both the communication and computation costs for ObjStore-Agg and computation costs for FLStore where communication costs are negligible. We can observe that the majority of the cost stems from the I/O (including communication) of data relevant to the non-training workloads. Resnet18, EfficientNet, and MobileNet spend 87.46%, 76.96%, and 85.80% of their total time respectively in I/O, and SwinTransformer spends 53.32% percent

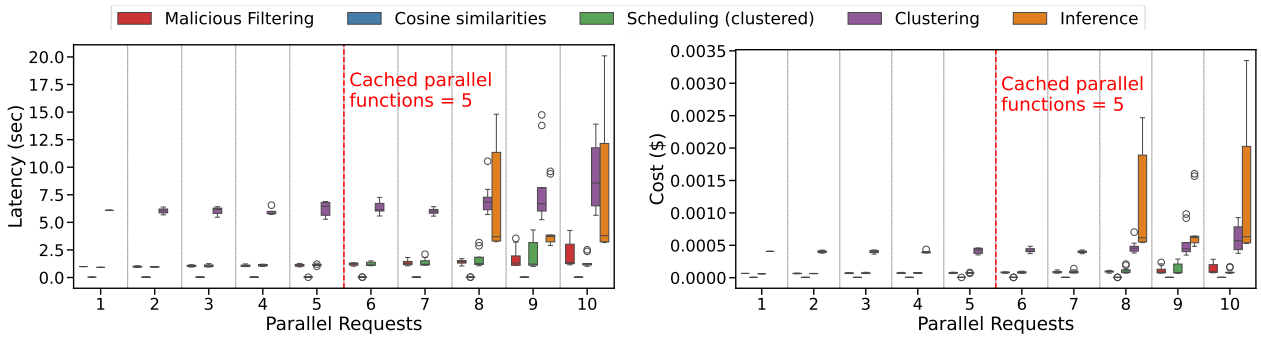

Figure 11. FLStore **scalability** for iteratively increasing parallel requests and 5 parallel cached functions.

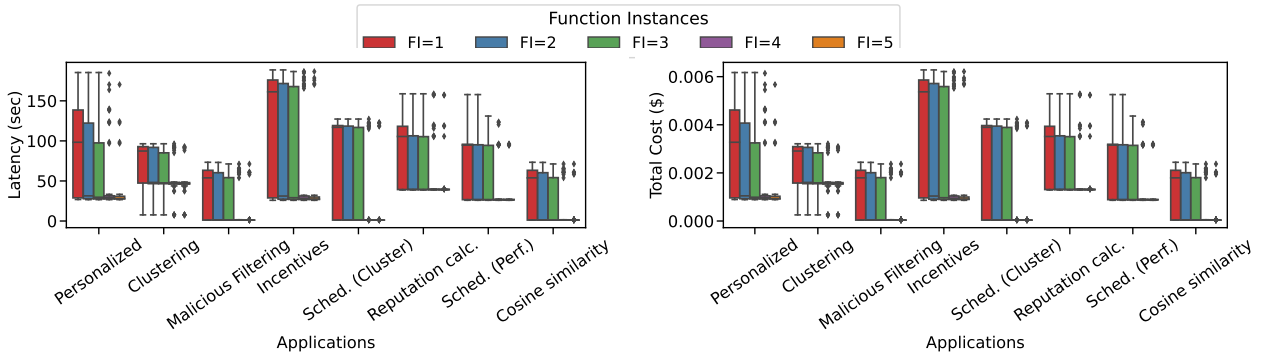

Figure 12. FLStore latency and cost per request over 50 hours with varying function instances (FI) for **fault tolerance**.

of its total time in I/O. Thus, by reducing the I/O time and data transfer costs FLStore provides a cost-effective solution for offloading the non-training workloads in FL. Across 50 hours and 3000 total requests we see that Resnet18, MobileNet, and EfficientNet show a 94.73%, 92.72%, and 86.81% average decrease in cost respectively, and Swin-Transformer has an average 77.83% reduction in cost.

B.2 FLStore vs In-Memory Cache

We also perform the total cost breakup analysis over 50 hours, 3000 total non-training requests, and 10 workloads, calculating both the communication and computation costs for Cache-Agg and FLStore. Results for this analysis are shown in Figure 16 FLStore decreases the total time by $37.77\% - 84.45\%$ amounting to 191.65 accumulated hours reduced for all requests and a $98.12\% - 99.89\%$ decrease in total cost resulting in a reduction of $\$7047.16$ accumulated dollar costs for all 3000 requests across 50 hours. To compare both (Cache-Agg and ObjStore-Agg) on the same workloads tested with Cache-Agg, FLStore shows an average decrease in latency of 71% with ObjStore-Agg and 64% with Cache-Agg, the decreases with ObjStore-Agg is larger as cloud object stores are slower than cloud caches. However, in terms of costs cloud caches are more expensive than cloud object stores, which is why for the workloads tested with Cache-Agg, FLStore shows an average decrease in costs of 98.83% compared to Cache-Agg and 92.45% decrease compared to ObjStore-Agg.

B.3 Overall cost reduction with FLStore

We also evaluated the overall reduction in FL costs brought by optimizing non-training workloads through FLStore. Figure 17 shows that although FLStore does not directly reduce training costs, it significantly lowers the overall per-round cost by minimizing the communication overhead associated with non-training tasks. For example, debugging costs are reduced from $\$0.099$ to $\$0.004$ (a reduction of 96.4%), and inference costs drop from $\$0.097$ to $\$0.004$ (a 96% reduction). Other tasks, such as reputation calculation and cosine similarity, also see substantial cost savings. These findings show that FLStore eliminates the need for frequent data transfers that typically inflate non-training costs also reducing the overall cost of FL jobs.

C FLStore Static: Ablation Study

For comparison with FLStore-Static, we consider a scenario where the workload changes from *model inference* to *malicious filtering*. Caching policy of FLStore-Static remains static (Individual Client Updates) which was for model in-

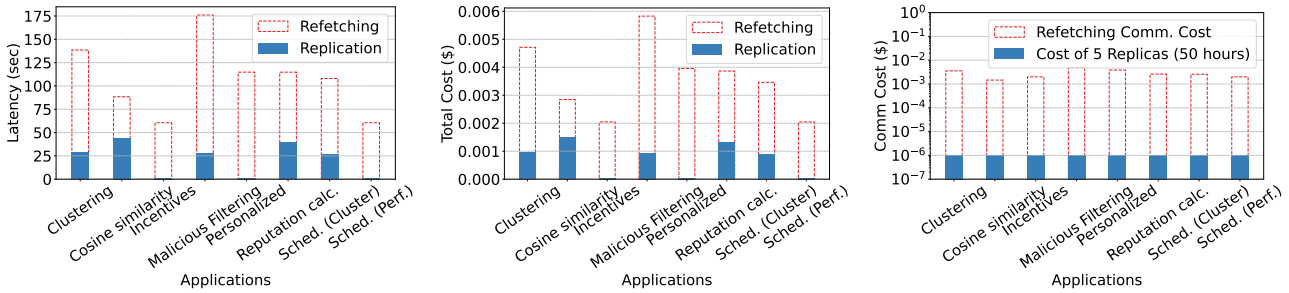

Figure 13. Overall latency and cost comparison of replication vs. re-fetching (first and second from left), and communication cost comparison (rightmost).

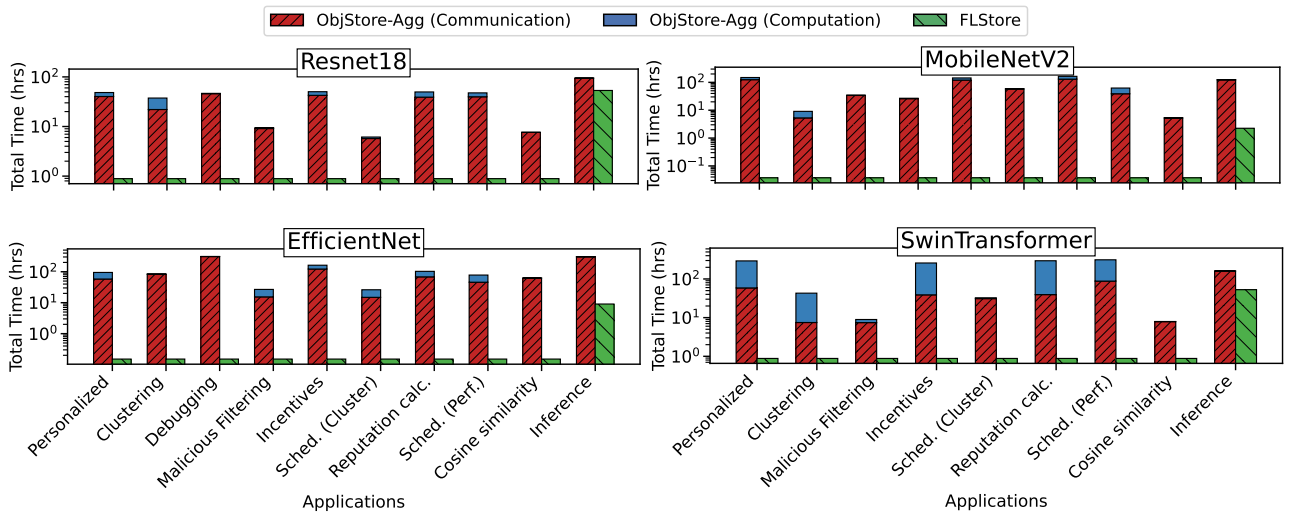

Figure 14. FLStore vs. ObjStore-Agg **total time** breakup comparison over 50 hours.

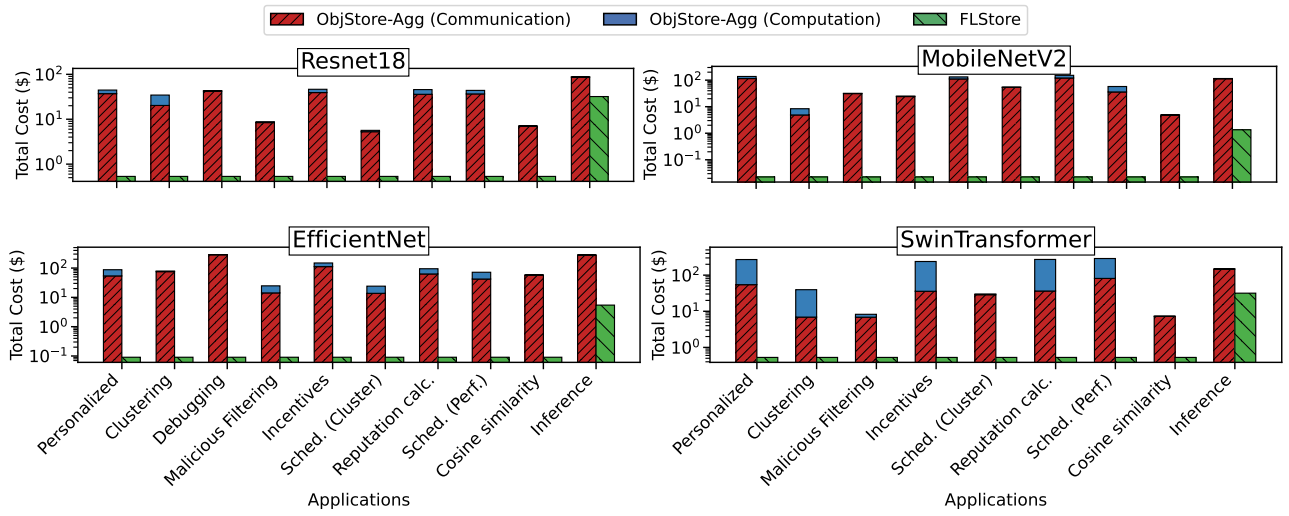

Figure 15. FLStore vs. ObjStore-Agg **total cost** breakup comparison over 50 hours.

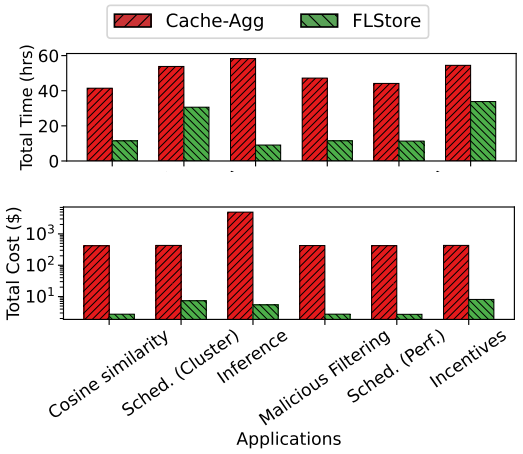

Figure 16. **Total time** (top) and **total cost** (bottom) comparison of Cache-Agg baseline vs. FLStore over 50 hours and 3000 total requests.

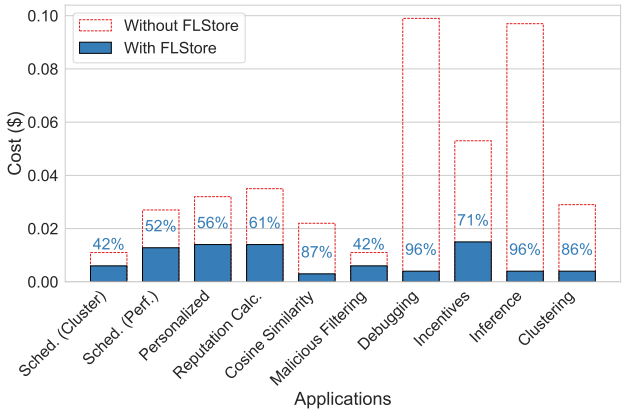

Figure 17. Overall cost of FL process per round with and without FLStore, with 200 clients, EfficientNet model (Tan & Le, 2021), 1000 training rounds, and CIFAR10 Dataset (Krizhevsky, 2009).

ference workload while FLStore changes its caching policy to *All Client Updates* based on the new workload (malicious filtering). Results in Figure 18 show that FLStore reduces per-request average latency by 99% (8 seconds) and costs by approximately 3×. This analysis highlights the importance of designing caching policies tailored for non-training FL workloads.

D Discussion: Limitations and Future Work

Support for Foundation Models Foundation Models are a class of models that have undergone training with a broad and general data set. Users can then fine-tune foundational models for specific use cases without training a model from scratch. We have added and evaluated several foundation

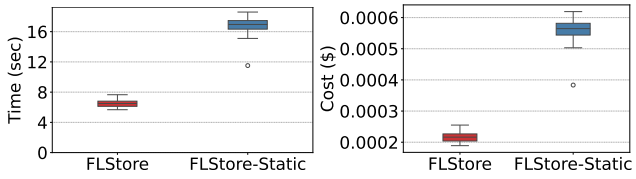

Figure 18. FLStore vs. FLStore-Static: **Per request latency** (left) and **cost** (right) while filtering malicious clients.

models from Figure 19 in FLStore and continue to add more such models. We also add model inference as an application for FLStore with the aim of providing a cost-effective alternative for serving models efficiently compared to other cloud solutions such as AWS SageMaker (Amazon Web Services, Inc., 2024b) which incurs high latency and costs as shown by our analysis in Figures 14 and 15.

FLStore Integration FLStore is built with a modular architecture that makes integration into any FL framework straightforward. We have integrated FLStore with IBMFL (IBM, 2020) and FLOWER (Beutel et al., 2020), both widely used in industry and research. FLStore includes key modules such as the Cache, Cache Engine, and Request Tracker. These modules are designed for portability and can be deployed using serverless platforms such as AWS Lambda. For closed-source platforms, networking between modules can be established using techniques such as reverse proxies similar to those used in InfiniCache (Wang et al., 2020). In simpler setups, the Cache Engine and Request Tracker can be deployed locally while caching is managed by serverless services like AWS Lambda. Alternatively, open-source platforms like OpenFaaS (Ellis & Contributors, 2024), which are natively supported by FLStore, provide even easier deployment options.

Adaptive Caching Policies Our ongoing efforts include designing agents based on Reinforcement Learning with Human Feedback (RLHF) that can understand the characteristics of non-training workloads and create new caching policies for those workloads using our existing caching policies as a base. RLHF has successfully been deployed for hyperparameter and optimization configuration in FL (Khan et al., 2024b) and the configuration of caching policies is a similar challenge that we hope to resolve by employing this technique.

Function Memory Limitations Serverless functions are limited in memory resources having a maximum of 10 GB memory (Amazon Web Services, Inc., 2024a). This is more than sufficient for handling non-training workloads for cross-device FL even for small transformer models such as (Face, 2024) and Llama 3.2:1B (AI, 2024). As shown in Figure 19, the average size of popular models used in

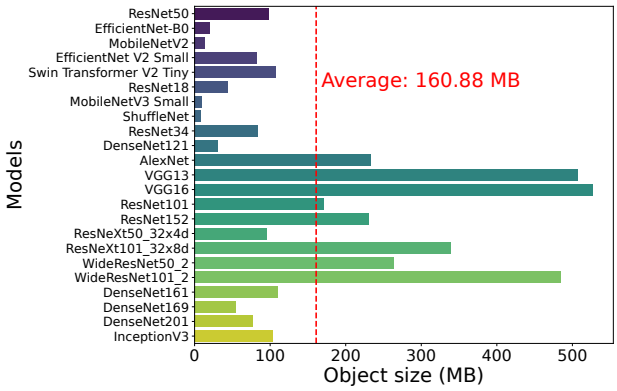

Figure 19. Memory footprint of commonly used models in FL.

cross-device FL is just 161 MB approximately. However, for even larger foundational models such as Large Language Models (LLMs) with greater than 1B parameters (Brown et al., 2020), we are working on utilizing pipeline parallel processing where function groups can be assigned for each workload and each function in that group can perform computations in a pipeline parallel manner (Jiang et al., 2021; Yang et al., 2022b).

Cached data types FLStore's data plane supports a flexible set of data types central to FL. Typically, it caches model weights, metadata (including versioning and training configurations), and the training/validation data itself. In some cases, additional data such as intermediate activations are stored to facilitate operations like recomputation or debugging in applications like FedNLR (Wang et al., 2024a) or FedDebug (Gill et al., 2023). FLStore can be used to store any type of data as singular objects less than 10 GB and as partitioned objects greater than 10 GB, offering flexibility for diverse FL workloads without the limitation of a fixed schema.

Training and Total FL Process Calculation Our paper provides a detailed analysis of the latency and cost components for non-training workloads. It is important to note that training latency and costs depend on the heterogeneous resources available on client devices. In many scenarios with hundreds of clients, only a few actively participate in training while non-training tasks can account for up to 97% of the overall FL latency. For controlled evaluation and meaningful comparison between training and non-training workloads, we assume that training is executed on a uniform compute environment. In this work, training is performed on the AWS SageMaker ml.m5.4xlarge instance described in §5.1. AWS SageMaker is widely used for edge training and inference tasks due to its scalable and flexible infrastructure, making it a realistic choice for federated learning evaluations (Amazon Web Services, 2023; Liberty et al., 2020; Das et al., 2020).

A Artifact Appendix

A.1 Abstract

FLStore is a serverless framework for efficient FL non-training workloads and storage. It unifies the data and compute planes on a serverless cache, enabling locality-aware execution via tailored caching policies to reduce latency and costs. FLStore integrates seamlessly with existing FL frameworks with minimal modifications, while also being fault-tolerant and highly scalable. For efficient portability, we have implemented FLStore on top of OpenFaaS, which is an open-source serverless solution. In addition, we also provide containerized simulators that are easy to set up and can be used to evaluate the efficacy of FLStore in comparison to cloud-based solutions with disjoint compute and data planes. The code is available at `https://github.com/SamuelFountain/FLStore.git`.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Tailored caching policies for non-training FL workloads based on a workload taxonomy (P1–P4).
- **Program:** FLStore framework implemented as serverless functions using OpenFaaS, along with Kubernetes-based containerized simulators.
- **Compilation:** Docker-based container images built via provided Dockerfiles.
- **Transformations:** Caching transformations including prefetching and eviction strategies tuned to iterative FL access patterns.
- **Binary:** Pre-built container images available in the repository.
- **Data set:** Simulated FL non-training workload traces and public datasets (e.g., CIFAR10) used in evaluation experiments.
- **Run-time environment:** OpenFaaS, Serverless functions, cloud storage (MinIO/AWS S3), Docker.
- **Hardware:** Cloud-based serverless environment (e.g., AWS Lambda or equivalent) with typical function memory limits (up to 10 GB).
- **Run-time state:** Ephemeral state in serverless functions with persistent backup in MinIO.
- **Execution:** Containerized deployment on OpenFaaS orchestrated by provided scripts.
- **Metrics:** Per-request latency (communication and computation), cost per request, cache hit rate.
- **Output:** Logs, performance metrics, comparing FLStore against baseline cloud aggregators.

- **Experiments:** Latency and cost evaluations, scalability tests, and fault-tolerance experiments across multiple non-training workloads.
- **How much disk space required (approximately)?:** Approximately 10–15 GB (for container images and experimental data).
- **How much time is needed to prepare workflow (approximately)?:** 30–60 minutes for setup and configuration.
- **How much time is needed to complete experiments (approximately)?:** Reduced experiments can complete within 1–2 hours; full-scale evaluations may run over extended periods (up to 50 hours).
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Data licenses (if publicly available)?:** Public domain for benchmark datasets (e.g., CIFAR10); simulated data provided under the same license as the code.
- **Workflow framework used?:** OpenFaaS, Kubernetes.
- **Archived (provide DOI)?:** Archived on Zenodo (https://doi.org/10.5281/zenodo.14986611).

A.3 Description

A.3.1 How delivered

The artifact is delivered as a GitHub repository available at `https://github.com/SamuelFountain/FLStore.git`. The repository contains the full source code, Dockerfiles, deployment scripts for OpenFaaS, containerized simulators, and comprehensive documentation to reproduce the experiments.

A.3.2 Hardware dependencies

No specialized hardware is required. A machine capable of running Docker and accessing a cloud provider's serverless platform (or a Kubernetes cluster running OpenFaaS) is sufficient. We also have scripts that will help getting started quickly by deploying OpenFaas and MinIO on Kubernetes.

A.3.3 Software dependencies

- Docker and Docker Compose.
- Kubernetes (K3s).
- OpenFaaS (or an equivalent serverless function framework).
- Python 3.7 or later.
- MinIO (or an S3-compatible object store) for persistent storage.
- Standard command line tools (Git, Bash).
- Operating System: Ubuntu is preferred.

A.3.4 Data sets

The artifact includes simulated traces representing FL non-training workload patterns and uses public datasets (e.g., CIFAR10) for evaluation purposes. All simulated data is provided within the repository.

A.4 Installation

Steps to install and deploy FLStore:

1. Clone the repository from `https://github.com/SamuelFountain/FLStore.git`.
2. Install Docker and Docker Compose on your system.
3. Run the scripts according to the Getting Started instructions (in documentation) provided in the GitHub repository to install all dependencies and datasets.
4. Deploy the FLStore functions on OpenFaaS using the supplied deployment scripts.
5. Configure the persistent storage by setting up MinIO or linking to an S3-compatible service.

Detailed instructions are provided in the repository's README file.

A.5 Experiment workflow

Overview of the experimental workflow:

1. **Deployment:** Deploy FLStore on the OpenFaaS platform using the provided scripts.
2. **Simulation:** Launch containerized simulators to generate non-training FL workload requests. Parameters such as client count, training rounds, and workload types can be adjusted.
3. **Execution:** FLStore processes incoming requests with its tailored caching policies. In parallel, baseline systems (e.g., cloud object store aggregators and in-memory cache aggregators) are deployed for comparison.
4. **Data Collection:** System logs and performance metrics (latency, cost, cache hit rates) are collected and aggregated.
5. **Analysis:** Use the provided analysis scripts to generate graphs and tables comparing FLStore's performance with baseline setups.

A.6 Evaluation and expected result

The evaluation aims to demonstrate that FLStore:

- Reduces per-request latency by up to 50–70% compared to traditional cloud aggregators.
- Achieves substantial cost savings (up to 88–99% cost reduction) by minimizing communication overhead.
- Attains high cache hit rates (approximately 98–99%) due to its workload-specific caching policies.
- Scales effectively and maintains fault tolerance with minimal overhead.

Detailed performance graphs, numerical comparisons, and logs are included in the repository.

A.7 Experiment customization

Users can customize the experiments as follows:

- Adjust the number of simulated clients, training rounds, and non-training request patterns via configuration files.
- Switch between baseline configurations (cloud object store vs. FSLtore) to perform comparative studies.
- Extend the simulator to incorporate additional FL models or workload types.

A.8 Notes

- The artifact is provided as-is for research reproducibility purposes.
- Ensure that your serverless environment is properly configured to reflect realistic network conditions.
- Users are encouraged to report any issues via the GitHub repository's issue tracker.
- Results may vary depending on the underlying hardware and network conditions.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

The code is available at https://github.com/SamuelFountain/FLStore.git.