WIP: Pods: Privacy Compliant Scalable Decentralized Data Services

Jonas Spenger[1,2], Paris Carbone[1,2], and Philipp Haller[2]

[1] RISE Research Institutes of Sweden, Stockholm, Sweden
[2] Digital Futures and EECS, KTH Royal Institute of Technology, Stockholm, Sweden
{jonas.spenger, paris.carbone}@ri.se
{jspenger, parisc, phaller}@kth.se

Abstract. Modern data services need to meet application developers' demands in terms of scalability and resilience, and also support privacy regulations such as the EU's GDPR. We outline the main systems challenges of supporting data privacy regulations in the context of large-scale data services, and advocate for causal snapshot consistency to ensure application-level and privacy-level consistency. We present Pods, an extension to the dataflow model that allows external services to access snapshotted operator state directly, with built-in support for addressing the outlined privacy challenges, and summarize open questions and research directions.

Keywords: Decentralized Data Services, Dataflow Model, Privacy Compliance, GDPR.

1 Introduction

Implementing and maintaining distributed data services is becoming an increasingly complex task across two frontiers. At one end, there is strong demand for data decentralization across multiple data stores, scalability and improved resilience to failures [17,23,4]. At the other end, there is demand for user data protection and support for users to exercise their data protection rights [9,5]. Building large and complex data services over a single ACID (atomicity, consistency, isolation, and durability) database is no longer a realistic implementation approach for meeting today's demands [18]. Existing scalable solutions to building such services instead settle on weaker consistency models such as eventual consistency, which has become the norm for building large-scale data services.

In this work, we identify the core challenges that privacy regulations such as GDPR [9] and CCPA [5] add to the already existing set of requirements for building scalable data services, at the intersection of privacy-policy driven demands and systems driven demands. In particular, we argue that stronger types of consistency are required, and feasible to achieve given the necessary paradigm shift in modelling data services.

To that end, we propose Pods, a dataflow model that provides built-in support for consistent continuous processing of user data, as well as access to causally

snapshot consistent state such as materialized views used by external services. Our model is currently under implementation on top of the Akka actor framework [12] and features causal consistency for cross-state reads via the use of distributed consistent snapshotting [6], and the serializable execution of privacy requests. The solution supports ideas from recent positions on data-privacy protection systems [16,22,19] while expanding on the capabilities of modern dataflow systems [2,7], proposing stronger types of guarantees and ways of stateful processing relevant to data privacy.

In summary, we claim the following contributions: (1) We outline the main systems challenges for supporting data privacy regulations in the context of large-scale data services. (2) We argue that eventual consistency is insufficient for supporting privacy regulations and advocate the adoption of causal snapshot consistency, as implemented on dataflow systems. (3) We propose PODS, a system model capable of addressing all outlined challenges. (4) We summarize open questions and propose several research directions for resilient, scalable and privacy-protecting services on dataflow systems.

2 Problem Scope and Challenges

2.1 Privacy Regulation Preliminaries

Data privacy regulations such as the EU's General Data Protection Regulation (GDPR) [9] and the California Consumer Privacy Act (CCPA) [5] have shaped the landscape for data privacy conformance and the proper handling and protection of user data. The GDPR mainly concerns with how *controllers* (the data service providers) process and collect the data of *data subjects* (the users), and what rights the data subject has over its *personal data* (any data relating to an identifiable natural person). The data subject may issue *privacy requests* (our notation) to the controller, these are requests to exercise the rights of the data subject. Evidently, enforcing compliance becomes more complex as data service architectures become decentralized. To illustrate this issue we focus on three fundamental data subject rights (*i.e.,* privacy requests) from the GDPR [9]:

1. Right of Access (Art. 15). The right of access grants the data subject access to information from the data service (controller) within one month's time on what personal data of the data subject is being processed, how it is being processed, the period for which the data will be stored, the purposes of the processing, the recipients of the processed data, and more.

2. Right to Erasure (Art. 17). The right to erasure grants the data subject the right to erase all personal data concerning the data subject within one month from the time of the request. This would include data that has been processed, and data for which there is no longer a legal ground for processing.

3. Right to Objection (Art. 21). The right to objection grants the data subject the right to object to certain types of processing if there are no legitimate grounds for the processing. Such a request should be processed within one month.

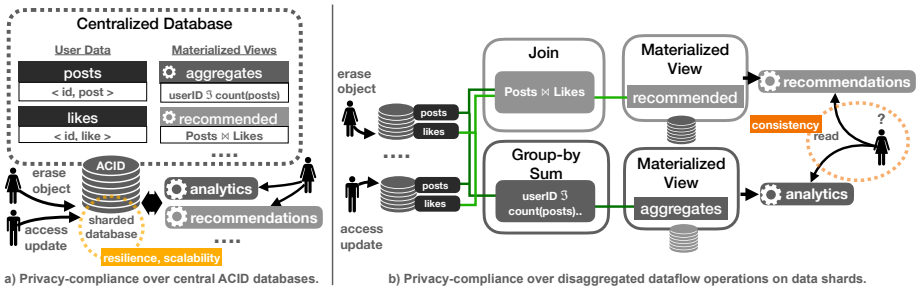

Fig. 1. Centralized and decentralized privacy compliant service composition.

2.2 Problem Intuition

Consistent Privacy Requests. From a data management systems perspective, granting the rights to perform privacy requests such as access, erasure and objection, can be considered as additional *operations* that need to be performed. To illustrate this, consider the example in Figure 1.a of a social network that records the posts and likes of users, and computes aggregates and recommendations based on these, which are used for an analytics and recommendations service, respectively. In the example the data subject issues an *erasure/access/objection* request to the central database management system (DBMS) with support for ACID (atomicity, consistency, isolation, durability) transactions. This request is committed transactionally with an immediate effect. In this setting, ACID transactions ensure trivially that the client and external services access a consistent view of completed operations and privacy requests.

In reality, however, most data services today are not built around a central DBMS with ACID guarantees. Instead, they employ decentralized storage and processing across geographically distributed data centers. The lack of support for ACID transactions in this setting makes it challenging to support the consistent execution of privacy requests (in contrast to regular user operations, privacy requests are expected to be executed with stronger guarantees). To that end, a dataflow-driven design has been proposed (exemplified in Fig. 1.b) for data privacy compliance by construction [16]. In this design, data services can be built organically from data shards, dataflow operators, and materialized views. Data shards are data sources owned by the users of the system. External services are composed using dataflow operators that subscribe to shards or other intermediate dataflow dependencies defined by other services, and end in materialized views. Data access by an external service is limited to reading from materialized views (*e.g.,* the recommended view in Fig. 1.b) that are composed on the fly through consumption of data events originating from the user shards. This is a promising architecture that aligns well with current trends in cloud computing. However, the current state of the art in distributed dataflow computing lacks two properties that we consider necessary for serving privacy requests, namely: causal consistency and serializability of privacy requests.

Dataflow causal consistency. Supporting materialized views in the distributed dataflow model is currently limited to eventual consistency [16] which is insufficient for serving privacy requests. To illustrate the problem consider the external observer in Fig. 1.b that accesses state from the recommendations service. Assume that an erasure request has been completed such that a user's post is no longer visible. Subsequently, the same observer reads state from the analytics service where the aggregate post count still includes the user's post (*i.e.,* the erasure request has not yet reached the aggregates view). Given that the erasure request was already observed in a prior access that precedes the second read, this exposes a causality violation. In practice, numerous causality violations can naturally occur in externally accessed dataflow graph state. For example, eventually consistent materialized views may roll-back due to failure recovery; and reading from different materialized views may be inconsistent as one view may contain the effects of a privacy request whereas the other view may not contain these effects. Instead, an external observer (user or external system) should only read causally consistent snapshotted state of completed operations. More specifically, if an observer performs two subsequent read requests, r_1 and r_2 with causal relationship $r_1 \prec r_2$ that observe two states s_1 and s_2, then there should be a causal relationship between the states $s_1 \preceq s_2$, such that the observed operations and privacy requests that yield the state s_1: $o_1, \ldots, o_n$, are a prefix of those that yield s_2: $o_1, \ldots, o_m$, with $n \leq m$.

Serializability of privacy requests. Whereas causal consistency addresses the order of which operations are observed externally, the internal execution order of dataflow operations, including privacy requests, is still subject to arbitrary stream alignment. For example, propagating events may be reordered if they are separated into two different streams, and later joined into a single stream, because the joined ordering can be an arbitrary interleaving of the two streams. Such a reordering of privacy requests could result into partially applied operations and therefore offer an inconsistent view of the system. To ensure the correct execution of privacy requests we also require them to be serializable. This means that for every privacy request p, all operations preceding it need to take effect before p, whereas all subsequent operations need to observe the effect of p. More formally, consider the sequence of operations $o_1, \ldots, o_{k-1}, p_k, o_{k+1} \ldots o_n$, and p_k is a privacy request, then the effect should be equivalent to an execution that executes and completes $o_1, \ldots, o_{k-1}$ before the privacy request p_k starts its execution, and $o_{k-1}, \ldots, o_n$ start execution after p_k completes.

Executing privacy requests. The serializability and causal-consistency describe the order in which the requests are to be executed. Yet, there is a need to materialize privacy requests on top of distributed dataflow operators. For example, an access request should produce the requested data and return it to the requester. For an erasure/objection request, the correct execution may be more complicated as the request modifies state. The dataflow operator needs to correctly update its own state, and also emit sufficient information to dependent dataflow operators such that they can perform the request accordingly.

2.3 Supporting Privacy on Dataflows: Challenges Overview

We have identified the need for dataflow systems to provide built-in support for privacy requests. A look into modern/popular dataflow streaming [7,1] as well as serverless programming [4,17,23] solutions used to build data services reveals fundamental design challenges for supporting privacy regulations. These include a lack of causally consistent externally queryable state support, and support for serializable transactions. To that end, we derive a set of challenges towards the creation of scalable distributed programming systems, able to support privacy requests (access, erasure, objection) consistently. Intuitively, there is a need to combine the programming flexibility of actor models with the support for ad-hoc external queries and transactional ACID guarantees of DBMSs and the end-to-end reliability and scalability of modern dataflow stream processing systems. Based on these intuitions we outline the following challenges. While a number of previous systems address one or more of the challenges, to the best of our knowledge no existing system addresses all challenges simultaneously.

C1 **Dataflow composition for high-performance data streaming:** providing the compositional construction of dataflow graphs and enabling high-performance data streaming.
C2 **Automated resilience to failures:** dealing with partial process and network failures that might occur throughout the execution of data services while maintaining exactly-once processing semantics.
C3 **Automated scaling of data services:** automatically and elastically scaling the system to meet increasing and decreasing load.
C4 **Snapshot consistent externally queryable state:** providing external services access to causally snapshot consistent state of dataflow operators.
C5 **Support for privacy requests and data ownership:** supporting serializable privacy requests, ensuring that users have control of and access to their raw and derived data.
C6 **Transparent handling of privacy requests:** The privacy requests should be handled transparently by the system. In effect, the application developer should not need to implement any logic for handling privacy requests.

3 Proposed Extensions to Dataflow Architecture

At a high level, PODS resembles most existing dataflow system models [7,2,13], supporting arbitrary stateful event logic, compositional subscription to event streams and pipelined task execution. Its main distinctions lie at the execution logic employed within its dataflow tasks, called pod tasks. A pod features two distinct components, one handling regular event input logic and another handling state operations. This grants PODS the flexibility to transparently employ all special yet necessary local actions that can collectively ensure global system properties such as serializability of privacy requests and dataflow causal consistency. In this section, we discuss its core design choices.

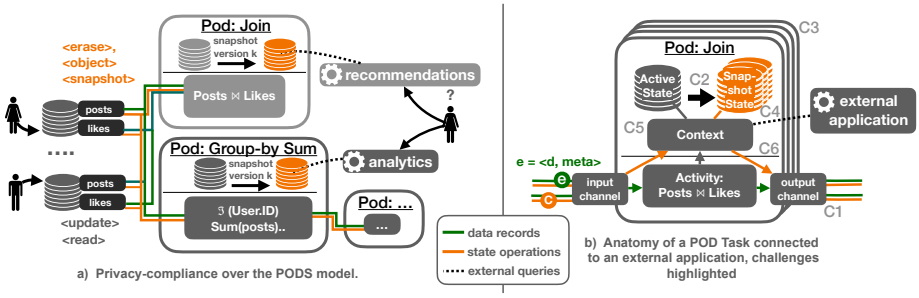

Fig. 2. Privacy compliant service composition with the PODS model.

3.1 Overview of the Pods Model

PODS is a dataflow model that processes user event streams and privacy requests. We adopt the notion of user shards [16] for all per-user data ingestion and introduce pod tasks for all stateful processing including the composition of materialized views. Figure 2.a shows an example of a privacy-compliant service in PODS, while Figure 2.b depicts the anatomy of a pod task. We further detail the design of user shards and pods, which constitute the overall behavior of the model.

User shards. All user data is ingested in "user shards" (adopted from [16]) that may be materialized on different data stores. A user shard creates a set of per-user data updates, e.g., new posts or likes as well as state requests including `erase`, `object` and system-invoked `snapshot` operations (Figure 2.a). All user shard streams are expected to be persistently logged and replicated. This makes them replayable and available in order to support rollback recovery.

Pod tasks. Pods execute the application and privacy request handling logic. In detail, pods: 1) subscribe to input streams and generate output streams; and 2) execute operations on the input stream events, and have two side effects: a) pod state is updated and b) new output events are generated. In contrast to existing dataflow models, PODS makes a clear separation between state and logic, one stateless control-flow component handles the application logic, and another stateful contextual component handles operations on state. Beyond this, pod tasks allow external services to query their snapshotted state.

We highlight the anatomy of a pod task in Fig. 2.b. This detailed view shows that a pod task consists of two components, a stateless *activity* component, and a stateful *context* component. Pod tasks are connected to other pods and user shards via a set of input and output channels. Events received by a pod on one of its input streams are processed sequentially, one at a time. Thus, the processing of a single event including its effects on the local state can be considered an atomic operation. Each input/output stream maintains a FIFO order of events; however, there is no deterministic ordering across streams. Application-level events are passed to the activity, the activity may access state via the *context* component, and emit messages on the output channels. Other events, such as

control events and privacy requests, are passed to the context component, which handles them accordingly (see Section 3.2). The PODS system attaches metadata to each data (events and state), such that it can derive the correct privacy policy [22] of raw and processed data using fine-grained information flow [21,15].

3.2 Handling Privacy Requests and State Management

The context manages two types of state, active state and snapshot state (see Figure 2.b). Active state is the live state of the executing system, and may be unstable as it has not been committed. The snapshot state of a pod task reflects its latest globally coordinated state snapshot of the dataflow graph and it can be used to support materialized views. Control operations, such as privacy request operations, are handled transparently by the context component (the control events are passed to the context component, not the activity), for which the context component may emit control events to other pods.

Both snapshot operations and privacy requests can make use of a similar broadcast and alignment dissemination scheme to enforce ordering. Similarly to classic marker-based snapshotting protocols (e.g., Chandy-Lamport [8]), markers can be inserted in dataflow inputs and further broadcasted to all outputs in order to separate those operations that precede and those that succeed a snapshot. To enforce the complete effect of certain operations an additional alignment phase is necessary [6]. The alignment makes every dataflow task prioritize pending changes across its inputs until all markers are received. This enforces all operations prior to a marker to complete before triggering a snapshot. Privacy requests can follow an identical broadcast and alignment scheme within a pod. This can enforce serializability for privacy requests.

External services can interact with the PODS system by querying the state of user shards and live pod tasks directly through an asynchronous RPC query (illustrated by the dotted lines in Figure 2.b). Updates on active pod state are not directly visible to external queries. This is because read operations would not expose the right level of isolation for external service access. Therefore, queries submitted by an external service receive the latest snapshotted version of that state from the pod context. Since snapshots within the pods dataflow are atomically committed across all pod tasks, subsequent external access requests would access the same or a newer version of the corresponding global state of the system. Via the use of a globally coordinated snapshotting method [6] it is guaranteed that an operation is either included in all pod snapshots, or pending to be committed in the next global snapshot.

Privacy requests that arrive at the pod are executed by the context component after the alignment phase. An access request can be executed on the local state, whereas erasure and objection requests are more difficult as they modify state. We can execute an erasure/objection request by using differential updates or by recomputing the state [16]. If these options are not available (*e.g.,* non-relational operators, user-defined functions), we can perform the operation directly on the state. The semantics and efficient/correct execution remains an open question which we intend to explore further (see Section 4).

Implementing a system with the presented properties efficiently is challenging. Whereas causal snapshot consistency has been shown to be supported in high-performance systems [6]; enforcing serializability of privacy user requests comes with an overhead from dissemination and synchronization of alignment markers. Further, certain privacy requests may cause large updates, which take a long time to complete. The total system overhead from privacy compliance, however, may be amortized through the batched execution of privacy requests, given that GDPR allows up to one month to process a privacy request.

3.3 Privacy Request Example

Let us revisit the example from Section 2 and exemplify how the PODS system can successfully deal with a privacy request (see Figure 2.a). Consider the `erasure` request submitted by the client. This request first arrives at a user shard. The shard can handle this request by erasing all data that belongs to the requesting client. We can find the corresponding data because of the information about the data origin contained in the metadata. This privacy request is then broadcast along all outgoing channels of the shard to the two other pod tasks (together with auxiliary information), the Join pod and the Group-by pod. The request arrives at the Join pod, and is passed to the context component which applies it to the active state and snapshot state. Similarly, the request is applied to the Group-by pod. If another pod subscribes to both the Join pod and the Group-by pod, the two streams are joined, and the privacy request will act as an alignment marker to ensure the serializability of the privacy request. This way, the request is propagated and applied consistently to the whole system.

3.4 Addressing the Outlined Privacy Dataflow Challenges

The specific features of the PODS model enable us to address the outlined challenges of privacy-compliant dataflows from Section 2.3, the challenges are highlighted in the Figure 2.b.

C1 Dataflow composition for high-performance data streaming. The PODS model enables constructing dataflow graphs composed of pod tasks interconnected via channels (see C1 in Figure 2.b). This design for high-performance data streaming is inspired to a large extent by state-of-the-art data streaming systems such as Apache Flink [7,6].

C2 Automated resilience to failures. Since PODS adopts the stateful stream processing paradigm and user shard streams are replayable, exactly-once stream processing methods such as distributed consistent snapshotting [6] and rollback recovery are applicable to ensure failure-recovery to a consistent active state.

C3 Automated scaling of data services. The PODS model enables scaling elastically according to load. A pod can either scale the number of messages that it can handle through executing messages concurrently on activities that are replicated across physical nodes, or scale its state by partitioning the state into shards across nodes (*e.g.,* using consistent hashing). This should be feasible

and requires little to no synchronization for activities that access disjoint state or conflict-free state (*e.g.,* keyed state [6]). Joint state has a synchronization overhead and may not be scalable, the developer could be notified of this through static checking and encouraged to use other types of state. The decision on when (policy) and how (mechanism) to elastically reconfigure [10] can be handled by the context component.

C4 Snapshot consistent externally queryable state. Updates on pod states are not directly visible to external queries. Instead, external queries are handled differently by the context components. All external read requests are granted access to the latest snapshotted state of a pod which is acquired via a global causally consistent checkpoint mechanism (see Section 3.2). This ensures that state is atomically committed across all pod tasks, and only completed operations are observable to external services.

C5 Support for privacy requests and data ownership. Supporting `access`, `erasure`, and `objection` requests requires us to be able to locate and update all data, raw or processed, belonging to or derived from a data subject. We can locate all data from a data subject by traversing the static dependencies of the dataflow system, and through the data ownership information in the metadata. Once the data has been located we can apply the requested operation on it. Privacy requests in PODS dataflows are directly related to the user state and not the application logic, thus, they are handled differently than regular application operations. Privacy requests broadcast to all outgoing channels (similar to snapshot markers), and aligned on pod tasks with multiple input streams (see Section 3.2). This way we can ensure a serializable ordering of the operations.

C6 Transparent handling of privacy requests. The application developer does not need to implement any logic for handling privacy requests, this is instead handled by the stateful context component within the pod task (see C6 in Figure 2.b). In effect, the application developer only needs to write the activity component's application logic, agnostic to any privacy/control logic.

4 Open Questions and Research Directions

The design for the PODS model presented in the previous section leaves a few questions unanswered, for which we outline research directions in the following.

A more flexible programming model. Supporting a wider range of scalable data services requires evolving dataflow graphs dynamically by adding and removing user shards and pod tasks at runtime. Generalizing pod tasks to actor-like entities could enable iterative computations which require cyclic data dependencies. However, cyclic data dependencies could conflict with assumptions that are critical for ensuring consistency, and the semantics of dynamic changes to the dataflow graph are still unclear. This research direction devises ways to efficiently ensure both consistency guarantees and privacy compliance in the presence of cyclic data dependencies and dynamic deployments.

Efficient information flow tracking. With the PODS model we propose to enable servicing privacy requests by employing fine-grained information-flow

tracking [21,15]. This poses an efficiency challenge for aggregate data. For example, computing aggregate data over all users of a system would have to track its origin to all users of the system. Efficient processing of such aggregate data would require a form of *declassification* [14] as it has been studied in the field of information-flow security. This research direction explores adaptations of declassification to enable efficient information flow tracking.

Execution of privacy requests. The execution of privacy requests has unclear semantics, for example, it is unclear how to handle a request on data associated to multiple users; and the efficient execution of privacy requests remains challenging [19]. There are various trade-offs between approaches depending on the workload. Further, many issues do not appear until implementing the full specification. This research direction looks at the efficient handling of fine-grained privacy requests for relational and user-defined functions, both for general workloads, and applied to specific case studies.

Consistent integration with external services. Data computed by a dataflow graph in the Pods model can be exported to external services which read from materialized views with snapshot consistency. Pull-based, snapshot-consistent reads have been presented in Section 3. However, it remains an open challenge (a) to support push-based updates and (b) to propagate privacy requests to external services with atomic consistency. Push-based updates pose a challenge due to the (strong) consistency on which external services should be able to rely. This research direction explores interfaces and protocols that enable atomically-consistent operations across dataflow graphs and external services, in order to provide end-to-end exactly-once data processing.

5 Related Work

Data privacy compliance. The Pods model was inspired by a position paper on "GDPR compliance by construction" by Schwarzkopf et al. [16]. In this work they propose a design that consists of user shards, a dataflow that computes on inputs from the user shards, and materialized views that are generated by the dataflow. Privacy requests are performed on the user shards, and these updates cause the dependent dataflow operators and materialized views to eventually update implicitly through the "partially-stateful dataflow model "[11]. In our model, we expose the pod state to external services, in replacement of materialized views, in order to provide causally consistent snapshot state of the system across views (such reads also access the metadata). The privacy requests are executed serializably using alignment markers; and we aim to support user-defined functions with the fine-grained information flow tracking, and declassification for aggregate data. Further, we adopt ideas from Data Capsules [22] to hold data together with metadata that specifies the policy of the data. The data capsule system consists of a data capsule manager that maintains the data capsule graph and tracks all data capsules, and verifies that analysis programs that access the data do not violate any policy. In the Pods system we enforce all data and events to be coupled with metadata (although our metadata is more limited), and have

no central manager as this information is decentralized. The MONPOLY system [3] uses logs to detect policy violation by formalising GDPR requirements into metric first-order temporal logic formulas. The GDPR for Akka Persistence supports encrypting data such that it later can be shredded by erasing the encryption key, this feature can be used to implement the right to be forgotten even when encrypted personal data may leak to logs.

Data services. Apache Flink [7] is a stream-processing framework for dataflow programs. Flink is known for the use of aligned snapshotting to achieve causal snapshot consistency [6]. PODS also builds on the same general dataflow model [7,2,13], and expands it further with the native alignment of privacy requests and view maintenance of all pod states. This new capability allows PODS to expose causally consistent shapshotted states for external reads featuring strong serializability of privacy requests. Flink Stateful Functions [20] runs on a runtime built on Apache Flink. Stateful functions are virtual, i.e. they don't consume any resources if idle, and the state and compute are separated. They provide built-in resilience in form of fault-tolerance and exactly-once semantics, and also support cyclical message patterns. Similarly, other serverless systems separate compute and state [4,17,23]. Durable Functions [4] provides serverless, elastic, failure-resilient, and consistent execution of workflows. It consists of orchestrations, i.e. reliable workflows, entities, i.e. actor-like addressable units, and critical sections, i.e. for synchronization. Cloudburst [17] is a function-as-a-service platform for stateful functions, for which state is held in a lattice-based distributed key-value store, and functions are executed in virtual machines with a local cache. Kappa [23] is a serverless computing framework that offers checkpointing for long-running tasks (and uses checkpointing for fault-tolerance). The outlined challenges are partially solved by these mentioned works, however, to the best of our knowledge, none of these projects provides built-in support for privacy compliance or for externally queryable snapshotted state.

6 Conclusion

We have presented PODS, a practical model for building scalable data services with privacy compliance as a core concern. Services in PODS can be built organically and execute reliably on decentralized infrastructures. To avoid inconsistencies between application-level operations and privacy requests, PODS employs transactional dataflow snapshots which capture a consistent view of the pod tasks' state, the snapshotting occurs asynchronously. PODS adopts best practices from distributed actor programming and serverless frameworks, which makes it flexible for supporting elastically scalable, replicated and fault tolerant services that respect their users' privacy by construction. The architecture of the PODS model allows the privacy request logic to be handled transparently, in effect the application developer is agnostic of any privacy logic.

Acknowledgements We would like to thank the anonymous reviewers for their helpful comments. This work was partially funded by the Swedish Foundation for Strategic Research (SSF grant no. BD15-0006) and by Digital Futures.

References

1. Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: Fault-tolerant stream processing at internet scale. Proc. VLDB Endow. **6**(11), 1033–1044 (2013). https://doi.org/10.14778/2536222.2536229, http://www.vldb.org/pvldb/vol6/p1033-akidau.pdf
2. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proc. VLDB Endow. **8**(12), 1792–1803 (2015). https://doi.org/10.14778/2824032.2824076, http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf
3. Arfelt, E., Basin, D.A., Debois, S.: Monitoring the GDPR. In: Sako, K., Schneider, S.A., Ryan, P.Y.A. (eds.) Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11735, pp. 681–699. Springer (2019). https://doi.org/10.1007/978-3-030-29959-0_33, https://doi.org/10.1007/978-3-030-29959-0_33
4. Burckhardt, S., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S.: Serverless workflows with durable functions and netherite. CoRR **abs/2103.00033** (2021), https://arxiv.org/abs/2103.00033
5. California Legislature: California consumer privacy act of 2018 (ccpa)) (2018), https://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.5
6. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in apache flink®: Consistent stateful distributed stream processing. Proc. VLDB Endow. **10**(12), 1718–1729 (2017). https://doi.org/10.14778/3137765.3137777, http://www.vldb.org/pvldb/vol10/p1718-carbone.pdf
7. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015), http://sites.computer.org/debull/A15dec/p28.pdf
8. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985). https://doi.org/10.1145/214451.214456, https://doi.org/10.1145/214451.214456
9. Council of the European Union: Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation) (2016), https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC
10. Fragkoulis, M., Carbone, P., Kalavri, V., Katsifodimos, A.: A survey on the evolution of stream processing systems. CoRR **abs/2008.00842** (2020), https://arxiv.org/abs/2008.00842

11. Gjengset, J., Schwarzkopf, M., Behrens, J., Araújo, L.T., Ek, M., Kohler, E., Kaashoek, M.F., Morris, R.T.: Noria: dynamic, partially-stateful data-flow for high-performance web applications. In: Arpaci-Dusseau, A.C., Voelker, G. (eds.) 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. pp. 213–231. USENIX Association (2018), https://www.usenix.org/conference/osdi18/presentation/gjengset
12. Lightbend, Inc.: Akka, https://akka.io/, accessed: 2021-05-21
13. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: Kaminsky, M., Dahlin, M. (eds.) ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013. pp. 439–455. ACM (2013). https://doi.org/10.1145/2517349.2522738, https://doi.org/10.1145/2517349.2522738
14. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: 18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France. pp. 255–269. IEEE Computer Society (2005). https://doi.org/10.1109/CSFW.2005.15, https://doi.org/10.1109/CSFW.2005.15
15. Salvaneschi, G., Köhler, M., Sokolowski, D., Haller, P., Erdweg, S., Mezini, M.: Language-integrated privacy-aware distributed queries. Proc. ACM Program. Lang. **3**(OOPSLA), 167:1–167:30 (2019). https://doi.org/10.1145/3360593, https://doi.org/10.1145/3360593
16. Schwarzkopf, M., Kohler, E., Kaashoek, M.F., Morris, R.T.: Position: GDPR compliance by construction. In: Gadepally, V., Mattson, T.G., Stonebraker, M., Wang, F., Luo, G., Laing, Y., Dubovitskaya, A. (eds.) Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2019 Workshops, Poly and DMAH, Los Angeles, CA, USA, August 30, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11721, pp. 39–53. Springer (2019). https://doi.org/10.1007/978-3-030-33752-0_3, https://doi.org/10.1007/978-3-030-33752-0_3
17. Sreekanti, V., Wu, C., Lin, X.C., Schleier-Smith, J., Gonzalez, J., Hellerstein, J.M., Tumanov, A.: Cloudburst: Stateful functions-as-a-service. Proc. VLDB Endow. **13**(11), 2438–2452 (2020), http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf
18. Stonebraker, M., Çetintemel, U.: "one size fits all": An idea whose time has come and gone. In: Aberer, K., Franklin, M.J., Nishio, S. (eds.) Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan. pp. 2–11. IEEE Computer Society (2005). https://doi.org/10.1109/ICDE.2005.1, https://doi.org/10.1109/ICDE.2005.1
19. Stonebraker, M., Mattson, T.G., Kraska, T., Gadepally, V.: Poly'19 workshop summary: GDPR. SIGMOD Rec. **49**(3), 55–58 (2020). https://doi.org/10.1145/3444831.3444842, https://doi.org/10.1145/3444831.3444842
20. The Apache Software Foundation: Apache flink stateful functions (2021), https://flink.apache.org/stateful-functions.html, accessed: 2021-06-14
21. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996). https://doi.org/10.3233/JCS-1996-42-304, https://doi.org/10.3233/JCS-1996-42-304
22. Wang, L., Near, J.P., Somani, N., Gao, P., Low, A., Dao, D., Song, D.: Data capsule: A new paradigm for automatic compliance with data privacy regulations. In: Gadepally, V., Mattson, T.G., Stonebraker, M., Wang, F., Luo, G., Laing, Y., Dubovitskaya, A. (eds.) Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2019 Workshops, Poly and DMAH, Los Angeles,

CA, USA, August 30, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11721, pp. 3–23. Springer (2019). https://doi.org/10.1007/978-3-030-33752-0_1, https://doi.org/10.1007/978-3-030-33752-0_1

23. Zhang, W., Fang, V., Panda, A., Shenker, S.: Kappa: a programming framework for serverless computing. In: Fonseca, R., Delimitrou, C., Ooi, B.C. (eds.) SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020. pp. 328–343. ACM (2020). https://doi.org/10.1145/3419111.3421277, https://doi.org/10.1145/3419111.3421277