Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows

Kyriakos Psarakis∗
Delft University of Technology
Delft, Netherlands
k.psarakis@tudelft.nl

Wouter Zorgdrager∗
Delivery Hero SE
Berlin, Germany
wouter.zorgdrager@deliveryhero.com

Marios Fragkoulis
Delivery Hero SE
Berlin, Germany
marios.fragkoulis@deliveryhero.com

Guido Salvaneschi
University of St. Gallen
St. Gallen, Switzerland
guido.salvaneschi@unisg.ch

Asterios Katsifodimos
Delft University of Technology
Delft, Netherlands
a.katsifodimos@tudelft.nl

ABSTRACT
Although the cloud has reached a state of robustness, the burden of using its resources falls on the shoulders of programmers who struggle to keep up with ever-growing cloud infrastructure services and abstractions. As a result, state management, scaling, operation, and failure management of scalable cloud applications, require disproportionately more effort than developing the applications’ actual business logic.

Our vision aims to raise the abstraction level for programming scalable cloud applications by compiling stateful entities – a programming model enabling imperative transactional programs authored in Python – into stateful streaming dataflows. We propose a compiler pipeline that analyzes the abstract syntax tree of stateful entities and transforms them into an intermediate representation based on stateful dataflow graphs. It then compiles that intermediate representation into different dataflow engines, leveraging their exactly-once message processing guarantees to prevent state or failure management primitives from “leaking” into the level of the programming model. Preliminary experiments with a proof of concept implementation show that despite program transformation and translation to dataflows, stateful entities can perform at sub-100ms latency even for transactional workloads.

1 INTRODUCTION
Organizations nowadays enjoy reduced costs and higher reliability, but cloud developers still struggle to manage infrastructure abstractions that leak through, in the application layer. As a result, managing application components, such as service invocation, messaging, and state management, require much more effort than the development of the application’s business logic [16]. Worse, moving a cloud application between cloud providers is prohibitive, due to significant differences in the underlying systems.

While there are multiple approaches for distributed application programming (e.g., Bloom [4], Hilda [50], Cloudburst [41], AWS Lambda, Azure Durable Functions, and Orleans [9, 12]), in practice developers mainly use libraries of popular general purpose languages such as Spring Boot in Java, and Flask in Python.

None of these approaches offers message processing guarantees, failing to support exactly-once processing: the ability of a system to reflect the changes of a message to the state exactly one time. Instead, they offer at-most- or at-least-once processing semantics. Programmers then have to "pollute“ their business logic with consistency checks, state rollbacks, timeouts, retries, and idempotency [32, 34].

We argue that no matter how we approach cloud programming unless an execution engine can offer exactly-once processing guarantees so that it can be assumed at the level of the programming model, we will never remove the burden of distributed systems aspects from programmers. To the best of our knowledge, the only systems able to guarantee exactly-once message processing [13, 39] at the time of writing, are batch [1, 20, 51] and streaming [14, 36, 45] dataflow systems. However, their programming model follows the paradigm of functional dataflow APIs which are cumbersome to use and require training, and heavy rewrites of the typical imperative code that developers prefer to use for expressing application logic.

For these reasons, we argue that the dataflow model should be used as a low-level intermediate representation (IR) for the modeling and execution of distributed applications, but not as a programmer-facing model. Technically, one of the main challenges in adopting a dataflow-based IR is that the dataflow model is essentially functional, with immutable values being propagated across operators that typically do not share a global state. Hence, adopting a dataflow-based IR entails translating (arbitrary) imperative code into the functional style. Compiler research has systematically explored only the opposite direction: to compile code in functional programming languages into a representation that is executable on imperative architectures – like virtually all modern microprocessors. Yet, the translation from imperative to functional or dataflow programming remains largely unexplored.

This paper presents a prototypical programming model, compiler pipeline, and IR that compiles imperative, transactional object-oriented applications into distributed dataflow graphs and executes them on existing dataflow systems. Instead of designing an external Domain-Specific Language (DSL) for our needs, we opted for an internal DSL embedded in Python - a language that is already popular for cloud programming. Specifically, a given Python program is first compiled into an IR, an enriched stateful dataflow graph that is independent of the target execution engine. That dataflow graph can then be compiled and deployed to a variety of distributed systems. The current set of supported systems includes Apache Flink Statefun and StateFlow – our own dataflow system built for the needs of such low-latency cloud
applications. The choice of a runtime system is completely independent of the application layer, which allows switching to different runtime systems with no changes to the application code.

The contributions of this paper go as follows:

- To the best of our knowledge, this is the first work to propose compiling and executing imperative programs into distributed, stateful streaming dataflows.
- We present a compiler pipeline that analyzes an object-oriented application and transforms it into an IR tailored to stateful dataflow systems.
- We describe an IR for cloud applications and how that IR translates to a dataflow execution graph, targeting a variety of distributed systems, thereby making cloud applications portable across different systems and infrastructures.
- We compare Stateflow, a novel transactional dataflow system, against Apache Flink Statefun and demonstrate the limitations of existing dataflow systems, motivating further research. Our experimental evaluation shows that Stateflow incurs low latency in the YCSB+T [22] workload.
- Despite the promising early results, several research questions remain open. We detail those in this paper and lay out a future research agenda.

The proposed system presented in this paper can be found at: https://github.com/delftdata/stateflow. A preliminary version of this paper is included as an abstract in CIDR 2023 [37].

2 FROM IMPERATIVE CODE TO DATAFLOWS

Historically, imperative programming and functional programming have evolved in parallel: imperative as a direct codification of (operational) computational models (e.g., Von Neumann architecture, Turing machines), and functional inspired by mathematical abstractions (e.g., lambda calculus, program denotation). While functional programming has been embraced by a number of languages (e.g., Haskell [29], ML [27]) imperative programming has taken the scene, with most mainstream languages featuring object-oriented (mutable) abstractions. Over the last years, imperative languages like Java and Python, which support a large variety of domain-specific packages, e.g., networking, statistics, numeric computation, etc. have become extremely popular among non-expert programmers.

Yet, the benefits of functional programming have been known for a while. Most notably, functional code is often embarrassingly parallelizable because of the lack of side effects and mutability. Developers working with imperative languages – let alone non-expert developers – can hardly access this feature.

2.1 Approach Overview

The main principle behind our compiler pipeline is that developers simply annotate Python classes with @stateflow and the system automatically analyzes and transforms these classes into an intermediate representation which is then transformed into stateful dataflow graphs, ready to be deployed on a dataflow system. Similar to (Virtual) Actors [12, 49], entities can make calls to methods of other entities. Figure 1 depicts two sample entities: User and Item. Details of the programming model are provided in Section 2.2.

In the first pass of an Abstract Syntax Tree (AST) static analysis, we extract the class’s variables (i.e., instance attributes referenced with self/), the names of each method, and all respective types indicated by the programmer (Section 2.2). In the second round of analysis, classes that interact with each other are identified in order to create a function call graph (Section 2.3). Then the call graph is analyzed to identify calls to other functions (possibly residing in a remote machine), at which point functions have to be split, composing the final dataflow (Section 2.4).

This dataflow graph enriched with the compiled classes, execution plans, and all metadata obtained from static analysis comprises the intermediate representation (Section 2.5). Finally, that intermediate representation can be translated, deployed, and executed in different target systems (Section 3).

Note that a complete account of the analysis and transformation algorithms is not possible due to space limitations, but it will be provided in the extended version of this paper.
the incoming event. Since operators can be partitioned across multiple cluster nodes, each partition stores a set of stateful entities indexed by their unique key. When a function of an entity is invoked, the entity’s state is retrieved from the local operator state. Then, the function is executed using the arguments found in the incoming event that triggered the call, as well as the state of the entity at the moment that the function is called.

**Example.** A User operator as seen in Figure 2, is partitioned on username. Upon invocation of a function of the User entity, an event is sent to the dataflow graph’s input queues. The incoming event is partitioned on username by an ingress router. Via the dataflow graph, the event ends up at the operator storing the state for that specific User. The system then reconstructs the User object using the operator’s code and the function’s state and executes the function. Finally, the function return value is encoded in an outgoing event which is forwarded to the egress router. This egress router determines if the event can be sent back to the client (caller outside the system, such as an HTTP endpoint) or needs to loop back into the dataflow in order to call another function.

The need for function splitting. For simple functions that do not call other remote functions, both the translation to dataflows and the execution is straightforward. However, if the function `User.buy_item` calls the (remote) function `item.update_stock` whose state lies on a different partition, the situation becomes more complicated. Note that a streaming dataflow should never stop and wait for a remote function to complete and return before moving on with processing the next event. Instead, it must “suspend” the execution of e.g., `buy_item` of Figure 1, right at the spot that the remote function `item.price()` is called until the remote function is executed and an event comes back to the User operator with a return value.

In order to do this, we adopt a technique to transform the imperative functions into the continuation passing style (CPS) [38]. More specifically, we propose an approach to split a function definition into multiple ones (Section 2.4) at the AST level as depicted (approximately) in Figure 1.

### 2.4 From Imperative Functions to Dataflows

**References to Remote Functions.** After the first round of static analysis, the compiler identifies if a function definition has references to a remote stateful entity using Python type annotations. These functions may require function splitting. The algorithm traverses the statements of a function definition and the function is split either when a remote call occurs or on a control-flow structure. For example, the following `buy_item` calls the remote function `item.update_stock`:

```python
def buy_item(self, amount: int, item: Item):
    total_price = amount * item.price
    is_removed = item.update_stock(amount)
    return total_price
```

This function is split at the assign statement on line 3 and results in two new function definitions:

```python
def buy_item(self, amount: int, item: Item):
    total_price = amount * item.price
    return total_price
```

```python
def update_stock(arg):
    return total_price,
```

```python
def buy_item(self, total_price, update_stock_return):
    is_removed = update_stock_return
    return total_price
```
The `buy_item_0` function defines the first part of the original function and it evaluates the arguments for the remote call. The `buy_item_1` function assumes the remote call `item.update_stock` has been executed and its return variable is passed as an argument. In general, each function that was split takes as arguments the variables it references in its body and returns the variables it defines. For example, since `buy_item_0` defines the variable `total_price`, its value is returned from the function. Next, since `buy_item_1` uses `total_price`, it is defined as a parameter.

**Control Flow.** The compiler also needs to split functions when encountering remote function calls within control flow constructs like `if`-statements or `for`-loops. In short, an `if`-statement is split into three new definitions: one that evaluates its conditional, one that evaluates the 'true' path, and one that evaluates the 'false' path. Similarly, a `for`-loop is split into three new definitions: one that evaluates the iterable, one that evaluates the `for`-body path, and one that evaluates the code path after the loop. The function splitting algorithm is recursively applied to the statements inside the `for` path and inside the true and false path of the `if`-statement.

### 2.5 Intermediate Representation

Our intermediate representation is a stateful dataflow graph enriched with a number of aspects. After the static analysis, each dataflow operator is enriched with the entity/method names that it can run, their input/return types, as well as their method body. After splitting functions, we also need to build what we term a state machine. For every split function (Section 2.4), we maintain an execution graph that tracks the execution stage of a given stateful entity’s function invocation.

Essentially, the process of deriving the state machine consists of unrolling the control flow graph of the program. Conceptually, the translation to a state machine is possible by deriving a finite representation of the program. To this end, we i) do not allow unbounded recursion and we ii) keep track of the current iteration for loop control structures, by enriching the state machine with the additional state. When invoking a function that was split, the state machine is inserted into the function-calling event. As the event flows through the system, the execution graph is traversed and the proper functions are called. The execution graph stores intermediate results – the return values of the invoked functions.

### 3 RUNTIME DATAFLOW SYSTEMS

Stateful entities can be deployed as dataflow graphs to streaming dataflow systems, offering exactly-once fault-tolerance guarantees.

**Flink’s Statefun.** The IR is translated to a streaming dataflow graph that, for example, Apache Flink can execute. In that case, a Kafka source pushes events to the ingress router, which is a map operator performing a keyBy operation to route an event to the correct stateful map operator instance where function execution will take place. Each execution’s output is forwarded to the egress router, which forwards outputs to a Kafka sink.

We use Kafka to re-insert an event to the streaming dataflow, thereby avoiding cyclic dataflows, which are not supported by most streaming systems. Notably, our system implements all the logic required for routing and execution in this process. On the downside, when an event reenters a dataflow to reach the next function block of a split function, race conditions attributed to events coming from non-split functions could lead to state inconsistencies due to other events changing the same function’s state in the meantime. Time tracking with watermarks, support for cyclic dataflows, and locking could solve these problems. Since the IR is well-aligned with Statefun’s dataflow, only simple translation and mapping is required when using the Statefun runtime.

**StateFlow: a transactional dataflow system.** Existing dataflow systems cannot execute multi-partition transactions. To this end, we built StateFlow, a prototype dataflow system in Python. StateFlow treats each function – and the state effects it creates via calls to other functions – as a transaction with ACID guarantees. We achieve consistency by implementing an extension of Aria [35], a deterministic transaction protocol. The dataflow system is built to allow for dataflow cycles used in function-to-function communication and leverages co-routines for optimal resource utilization. For fault-tolerance StateFlow implements the consistent snapshots protocol [13, 15], which has been adopted by many streaming dataflow systems [5, 14, 30] alongside a replayable source as an ingress, allowing StateFlow to rollback messages and restore the snapshot upon failure. Although still a prototype, StateFlow is already able to execute transactional workloads (YCSB-T [22] and partly TPC-C) with promising performance (Section 4).

**Local.** A StateFlow dataflow graph can execute all its components in a local environment. The only difference is that the state is kept in a local HashMap data structure instead of a state management backend. Local execution allows developers to debug, unit test, and validate a StateFlow program as they would do for an arbitrary application. Afterward, they can simply deploy the program to one of the supported runtime systems.

### 4 PRELIMINARY EXPERIMENTS

For the experiments of this section, we opted for running Apache Flink Statefun against StateFlow (Section 3).

**Workload.** We are using workloads A and B from the original YCSB benchmark [18]. A is update-heavy – 50% reads 50% updates and B is ready-heavy – 95% reads 5% updates. In addition, we use the transactional workload T from YCSB+T [22], which atomically transfers an amount from one entity’s bank account to another (2 reads and 2 writes). For the throughput test, we defined a mixed workload M (45% reads 45% updates 10% transfers). For the latency tests, we use Zipfian and uniform key distributions.

**Setup.** We conducted all the experiments on 14 CPUs: 4 for the Kafka cluster, 6 for the systems, and 4 for the benchmark clients. For Statefun, we gave half of the resources to the Flink cluster and the other to the remote functions. StateFlow requires a single core coordinator, and the rest are used for its workers.

**Baseline.** In StateFlow, we execute complex business logic resulting in state operations. YCSB is a benchmark that supports simple inserts, deletes, and updates, not complete executions of transactions across multiple function calls. It is therefore expected for Stateflow, since it executes function calls and application logic, to have a larger overhead than key-value stores. StateFlow is not a key-value store; instead, it is a stateful function-as-a-service compiler and runtime that allows programmers to author object-oriented python code.

**Latency.** In the first experiment, we measured the end-to-end latency of all the YCSB workloads against the integrated backend systems with both Zipfian and uniform key distributions at the low amount of 100RPS. As seen in Figure 3 both systems perform well with low latencies across all workloads and distributions. Some interesting observations go as follows. First,
Statefun performs the same in both the A and B workloads and in both Zipfian and uniform distributions. This happens because Statefun does not use locking, allowing for concurrent access (but also inconsistency). Additionally, since all functions need to go to an external Python runtime, the cost of reads and writes are the same due to the network costs. We also observe that StateFlow outperforms Statefun because it allows for internal function-to-function communication and does not require the roundtrips to Kafka. Note that StateFlow additionally supports transactional workloads with higher latency than the rest but still, if we consider that a transfer operation is 2 read and 2 write operations, the transactional overhead of the system is minimal. Finally, we did not run Statefun against transactional workloads since it offers no support for transactions.

**Throughput.** In the second experiment, we gradually increase the input throughput and measure the end-to-end latency. This time we use the mixed workload that we defined, M (45% reads 45% updates 10% transfers). In Figure 4, we observe consistent results with the latency experiment up until the point where the difference in efficiency appears. The reason for this is that StateFlow is using more execution cores since it bundles execution, state, and messaging. In contrast, the Statefun deployment uses half its CPUs for messaging and state within the Apache Flink cluster and the other half for execution in a remote stateless function runtime. In the current experiments this balanced deployment was the optimal one in terms of resource utilization.

**System overhead.** Finally, we also measured the overhead that program translation (function splits, instrumentation, etc.) incurs as part of the complete runtime (not depicted for the sake of space preservation). We created a synthetic workload in which we varied different state sizes from 50 to 200kb. For each event, we measured the duration of different runtime components. Some of the components, like object construction, are attributed to program transformation overhead, whereas others, like state storage, are attributed to the runtime. In short, function splitting/instrumentation is only responsible for less than 1% of the total overhead.

**Conclusion.** The experimental evaluation demonstrates the potential of dataflows as an intermediate representation and execution target for scalable cloud applications. In short, these preliminary experiments show that we can translate imperative programs that hide all the aspects of distributed systems and error management from programmers and still achieve high performance. That said, the experiments also uncover the limitations of dataflow systems and implementation issues that we address in the following section.

### 5 OPEN PROBLEMS & OPPORTUNITIES

The ability to query the global state of a dataflow processor, as well as perform transactional state updates on its state, can transform a dataflow processor into a full-fledged, distributed database system. The envisioned system will be capable of executing Turing-complete "stored procedures" (such as the entity functions in the case of this paper) that are distributed, partitioned and can perform function-to-function calls with exactly-once guarantees. This is the ultimate goal of this work.

In this section, we discuss a number of opportunities emerging mainly from transactional workloads with low-latency requirements and outline future research directions to enable the adoption of dataflow systems for executing general cloud applications.

**Program Analysis.** The dataflow model is essentially a finite state machine where nodes are the functions from the original (Turing-complete) program and arcs indicate event flow. In the case of loops, events also carry information about the previous iterations of the loop (e.g., the variables that are read and written in the loop body and in the loop condition clause). This information handles loops correctly (Section 2.5). For method calls, if a method is mapped to a single state, it would be problematic to determine where to return after a call if in the codebase there are multiple calls that have different return points. We map each method call into a transition to a state that is specific for that call. This means that calls to the same method may result in a different state in the automata, ensuring that each of these states has as a next state the correct return point. This approach requires to unroll the program, expanding each potential method call that may occur at runtime into a different state.

Following this approach, recursive functions would result in a state for each recursive step. Since unbounded recursion would result in infinite automata, we prohibit recursion. Yet, from a compiler perspective, since a program can be CPS-transformed, recursion can be translated into loops via tail-call elimination [8], which could potentially affect the dataflow engine’s performance.

In addition, in what is traditionally referred to as dataflow languages (e.g., Esterel [11], Lucid [47]), the computation is driven by data propagation – just like in streaming dataflows. However, the expressivity of such languages has been intentionally limited to enable efficient execution (automatic) verification techniques. While in this work we aim to target Turing-complete Python programs, the trade-off between expressivity, efficiency, and automatic verification is yet to be researched in the future.

**Transactions.** Current dataflow systems guarantee the consistency of single-event effects on a given key of the state. In order to support transactional executions across stateful entities, we
could employ single-shot transactions [31] or, like in our proto
typical dataflow system (Section 3), borrow ideas from deter-
ministic databases [2, 35, 44] for minimizing the coordination of
transactions. In practice, a large percentage of transactions can be expressed as single-shot transactions [43]; very popular databases such as Amazon’s DynamoDB [40] and VoltDB [42] support single-shot transactions. These ideas can define how a programming model can support patterns that have been adopted by practitioners in the last years, starting from SAGAs [25] and Try-Confirm-Cancel [28].

Exactly-once, Latency & External Systems. Exactly-once guar-
antees can incur high latency: the outputs of a dataflow only be-
come visible after an epoch terminates successfully [13]. Epoch
intervals cannot be too small because they would incur a high
overhead. However, one can leverage causal recovery [48] and de-
terminants [39] alongside replayable sinks in order to minimize
the latency within each epoch. The replayable sinks are required to
be able to retrieve determinants. However, at the border of a
system, i.e., when a message leaves the dataflow graph and is sent
to an external system, replayable sinks may be hard to assume. In
that case, one should make use of more traditional techniques for
deduplication (e.g., the common idempotence keys used in the
HTTP protocol). Under certain assumptions (deterministic com-
putations, persistent/replayable request queues, etc.), such idem-
potence keys can be generated automatically. However, this will
not be the case for a generic distributed application, which will
to have to generate, keep track of, check, and recycle unique iden-
tifiers to enforce the delivery of its output exactly-once. These
issues have not been studied enough in the context of distributed
databases, neither in models for cloud programming.

Querying Stateful Entities. In previous work [46], we have
shown that querying the global state of a dataflow processor can be,
not only efficient but can also come with certain correctness
guarantees. Some work on querying actors has already been
done in the context of Orleans [10]. However, querying (e.g.,
with SQL) a set of entities still poses a number of challenges,
especially with respect to the tradeoff between the freshness and
consistency of query results. To this end, we could borrow ideas
from RAMP (read-atomic) transactions [7] that match well the
execution model of transactions and read operations in stateful
entities.

6 RELATED WORK

The idea of democratizing distributed systems programming is not new. For instance, in [17], the authors mention that a combination of dataflows and reactivity would provide a good execution model for cloud applications. In this work, we share the same belief and build a prototype towards that direction.

Programming models. In the past, approaches like Distributed
ML [33], Smalltalk [21], and Erlang [6] aimed at simplifying the
programming and deployment of distributed applications. Many
of those ideas, including the Actor model, can be reused and
extended today. Erlang implemented a flavor of the actor model.
Akka [49] offers a low-level programming model for actors. Clos-
est to our work is the Virtual Actors model introduced by Orleans
[9, 12], which aims at simplifying Cloud programming and even
supports some form of transactions [23]. However, Orleans re-
quires a specialized runtime system for virtual actors, which does not
support exactly-once messaging and does not compile its ac-
tors into stateful dataflows. Nonetheless, our work is heavily
inspired both by Orleans and by Pat Helland’s entities [28].

Imperative programming to Dataflows. The idea of trans-
slating imperative code to dataflow is not new. In the database
community, there has been work on detecting imperative parts of
general applications that can be converted into SQL queries (e.g., [24]) but also for automatic parallelization of imperative
code in multi-core systems. For instance, the work by Gupta and
Sohi [26] compiles sequential imperative code to dataflow pro-
grams and executes them in parallel. Our work draws inspiration
from both these lines of work and extends them by taking into
account the partitioning of state as well as other considerations
that we outline in Section 5.

Stateful Functions. A new breed of systems marketed as state-
ful functions such as Cloudburst [41], Lightbend’s Cloudstate.io
and Apache Flink’s Statefun.io [19], as well as our early pro-
totype in Scala [3] also aim at abstracting away the details of
deployment and scalability. However, none of those compiles
general-purpose object-oriented code into dataflows.

7 CONCLUSIONS

In this vision paper we argue that if we want to hide failures from
the top-level programming models of Cloud applications, exactly-
one guarantees should become a first-class citizen. While dataflow
systems can provide such guarantees, their programming model makes
the development of general Cloud applications cumbersome.
To this end, we have developed a compiler pipeline that
statically analyzes an object-oriented Python application in order
to create an intermediate representation in the form of a dataflow
graph, and then submit that dataflow graph to existing dataflow
systems. Leveraging dataflow systems’ exactly-once guarantees
can essentially hide all Cloud failures from programmers with low
overhead: our preliminary experimental evaluation demonstrates
that function splitting and program transformation incur less
than 1% overhead and the YCSB+T benchmark, with low-latency
execution.

Current Status. Despite the encouraging results, lots of prob-
lems remain open: specifically in the area of transaction exec-
cution, programming models, program analysis, and dataflow
engines for general cloud applications. Our work currently fo-
cuses primarily on i) strengthening the formal underpinnings of
program transformation to dataflows, ii) extending the pro-
gramming model with different transactional paradigms, and iii)
further developing StateFlow, our novel transactional dataflow
system.

ACKNOWLEDGMENTS

This publication is part of project number 19708, of the Vidi
research program which is partly financed by the Dutch Research
Council (NWO).

In memory of Eelco Visser.

REFERENCES

[1] [n.d.]. Apache Spark project. http://spark.apache.org/.
[2] Daniel J. Abadi and Jose M. Faleiro. 2018. An Overview of Deterministic
Database Systems. In Commun. ACM.
[3] Adil Akhter, Marinos Fragioukos, and Asterios Katifodimos. 2019. Stateful
functions as a service in action. In VLDB.
[4] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M Heller-
stein, and Russell Sears. 2010. Boom analytics: exploring data-centric, declara-
tive programming for the cloud. In EuroSys.
