Cascade: From Imperative Code to Stateful Dataflows

Marcus Schutte
Delft University of Technology
m.schutte@tudelft.nl

Lucas Van Mol
Delft University of Technology
l.vanmol@student.tudelft.nl

George Christodoulou
Delft University of Technology
g.c.christodoulou@tudelft.nl

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

ABSTRACT

Executing applications in the cloud is becoming increasingly popular, primarily developed as microservices containing imperative code. In our previous work, we have made the case that such applications can benefit from using dataflow-based runtimes in a cloud environment. In particular, dataflow-based runtimes offer significant advantages over imperative code, namely, exactly-once processing, transparent message handling, and coarse-grained fault tolerance offered by dataflow systems. However, dataflow programming is not preferred by developers. In this work we bridge this gap, namely, we present our progress towards creating a suitable intermediate representation (IR) that can be used to compile stateful imperative code into dataflows, enabling seamless migration to the cloud. We then present a compiler pipeline prototype that offers two key benefits: *i*) it enables program optimizations and data parallelism, and *ii*) it decouples the input program from the target execution environment, while allowing interesting optimizations. Preliminary experiments demonstrate that our IR optimizations speed up the p50 request latency by 267x on average.

ACM Reference Format:
Marcus Schutte, Lucas Van Mol, George Christodoulou, and Asterios Katsifodimos. 2025. Cascade: From Imperative Code to Stateful Dataflows. In *The 19th International Symposium on Database Programming Languages (DBPL '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3735106.3736537

1 INTRODUCTION

Building scalable and consistent applications remains a challenge, especially with the multitude of cloud provider offerings and abstractions. While cloud platforms provide mechanisms for scaling workloads, they often lack strong execution guarantees. The Function-as-a-Service (FaaS) paradigm, e.g., offers flexibility, but shifts the burden of managing failures, consistency, and error handling to developers, polluting business logic with error-handling code. Even when correctness is ensured, system performance is compromised, mainly because of storage and execution disaggregation. Additionally, cloud providers' APIs are often proprietary and lack standardization, hindering migration across providers.

DBPL '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1919-6/2025/06
https://doi.org/10.1145/3735106.3736537

A key challenge in distributed system programming lies in managing state and messaging, particularly in environments with commodity hardware prone to failure. Cloud computing solutions often combine FaaS with external object stores (e.g., Amazon S3) for state management. This approach introduces significant communication overhead, as data has to be loaded, modified, and sent over the network, an inefficiency known as "shipping data to code."

To mitigate these inefficiencies, solutions such as Cloudburst [39], Boom [9], Durable Functions [15], and Hilda [44] have emerged, improving cloud programming abstractions by integrating state management with computation. Actor-based systems like Orleans [14] provide structured models for distributed programming, but lack strong guarantees, such as exactly-once processing. Dataflow systems such as Flink [19] and Spark Streaming [10] express computation as directed graphs, optimizing data-parallel workloads but largely remaining within the domain of data analytics. Similarly, machine learning frameworks such as TensorFlow [2] use dataflow-based execution to improve communication and shared state handling.

Despite these advancements, a programming abstraction for cloud applications remains elusive. The ideal solution should relieve developers from consistency concerns, facilitate portability across cloud environments, and optimize execution depending on the state accesses. A key step toward this goal, is the representation of communication as a dataflow graph, representing message-passing between computation steps. Using a dataflow-based execution model, enables minimization of communication overhead and efficient execution of stateful applications at scale.

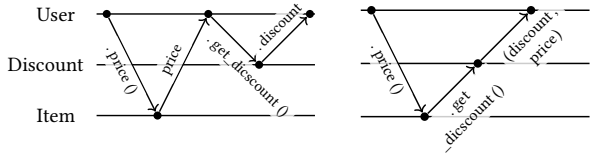
Motivating Example. To illustrate the challenges of distributed execution, consider a simple checkout process, as shown in Figure 1. The workflow starts with a request to the User entity, which retrieves an item from the user's basket (line 3). Since the Item entity is a separate component, potentially hosted on a different machine, fetching its price requires a network call (line 4). Once the price is obtained, the DiscountService determines any applicable discounts (line 5).

Traditional systems execute this sequence strictly step-by-step. As depicted in Figure 1b, each request and response flows through the User entity, effectively making it an orchestrator. In this design, correctness is ensured but results in high message complexity.

Our work is built upon the observation that messaging across services can be optimized by analyzing data flow. Instead of routing all calls through the User entity, we can allow the Item entity to communicate directly with DiscountService once the price is retrieved (Figure 1c). This approach resembles a "choreography", where components operate autonomously rather than relying on a

```python
class User:
    def checkout_item(self, discount_service: DiscountService):
        item: Item = self.get_item_from_basket()
        price: float = item.price()
        discount: float = discount_service.get_discount(price)
        # other checkout logic...
```

(a) Listing of a default implementation



(b) Orchestration **(c) Choreography**

Figure 1: Message complexity of an Orchestration compared to a Choreography.

central coordinator, usually referred to as call stream or promise pipeline [33]. As workflows become more complex, these optimizations yield even greater performance benefits.

Proposed Approach. This work builds on our previous research vision on dataflow intermediate representation [23, 36]. In this work, we propose Cascade, a compiler pipeline addressing the challenge of obtaining an imperative program's dataflow representation efficiently. We begin by devising a target intermediate representation (IR) that explicitly encodes the dataflow between computation steps. The goal of the IR is to capture data dependencies and updates to the entity state. It acts as an abstraction layer, decoupling application logic from the underlying execution environment and enabling seamless deployment across multiple runtime systems. To ensure consistency and fault tolerance, we leverage execution frameworks with strong runtime guarantees, such as Apache Flink [19].

To achieve this IR, we develop a compiler responsible for translating imperative programs into dataflows seamlessly. We adopt the same programming model, *stateful entities*, which enables users to express cloud applications as Python classes that encapsulate and persist application state [23].

Finally, the dataflow representation enables several execution optimizations. Cascade leverages these via two concrete optimizations, pipelining and parallelizing the dataflow graph. Our approach enhances program performance in a distributed runtime while maintaining the original program semantics.

In summary, our contributions are threefold: *i*) a dataflow graph base IR that captures data dependencies and updates to the entity state, *ii*) a compiler that takes a stateful entity source program and translates this to the IR, *iii*) compiler optimizations that optimize that IR at the granularity of stateful entities. Our evaluation shows latency speedups of 13x to 518x in p50 latency and 13x to 57x in p99 latency.

2 BACKGROUND

In this section, we introduce the core concepts on which we built our compilation process: control flow graphs (CFGs), the dataflow model, and intermediate representations (IRs). First, we examine the structural differences between CFGs and dataflow graphs, highlighting how execution order and data dependencies differentiate each model. Note that dataflow graphs are discussed here in the context of streaming data systems, not data dependency analysis. Building on this, we outline an approach for translating a CFG into a dataflow graph, leveraging data analysis techniques to extract dependencies between code fragments. However, the main focus of our work is handling state during transformation. As we detail in §3, effectively managing state is key to preserving correctness and ensuring an efficient compilation process.

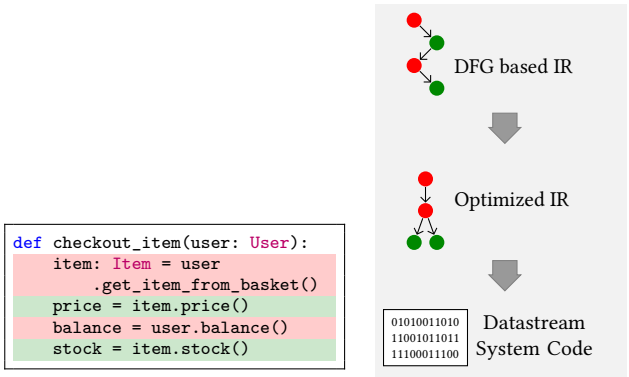
Intermediate Representation. Compilers use Intermediate Representation (IR) as a middle layer to translate source code into machine-executable code [3, 22]. IRs act as an abstraction layer, enabling optimizations, transformations, and code analysis [40]. IRs serve as a machine- and language-independent bridge between source code and target architectures. Multiple languages can compile into a common IR, which is then translated into different architectures. A notable example is LLVM IR [46].

Furthermore, IRs are formed to fit the usecase. For instance, Control Flow Graphs (CFGs) [3] aid control flow optimizations, while Static Single Assignment (SSA) simplifies data dependency analysis [45]. Modern compilers often use multiple IRs at different stages to enhance performance [41].

Optimizations. Compilers can optimize code by identifying Instruction-Level Parallelism (ILP) and Data-Level Parallelism (DLP) [29]. ILP exploits the absence of data dependencies and control hazards to execute multiple instructions in parallel within a processor pipeline. DLP applies the same operation to multiple data elements, often by analyzing loop bodies to detect independent iterations. In this work, we extend these techniques to stateful entities.

Control Flow Graph. A Control Flow Graph (CFG) [3, 7] is a directed graph where nodes represent basic blocks; code fragments, that are composed of a sequence of program statements that are executed sequentially. The edges indicate possible control flow paths between these blocks, illustrating how program execution progresses. An edge from block A to block B indicates that B may execute immediately after A. CFGs are building blocks in compiler design, program analysis, and software optimization by modeling loops, branches, and conditional structures. They facilitate static analysis techniques such as dead code elimination, reachability analysis, and vulnerability detection. A detailed algorithm for constructing a control flow graph can be found in [3].

Dataflow model. The dataflow model structures computation as a dataflow graph (DFG), where nodes represent operators: independent processing units, and edges define data exchange through message streams. Originally introduced as an alternative programming model for parallel computing [31], the dataflow model [4] has since been widely adopted in fault-tolerant distributed systems [1, 19, 38]. Dataflow programming eliminates the need for a program counter, as execution is triggered as soon as the required data becomes available. This enables automatic exploitation of data parallelism. This model is leveraged for both batch and stream processing where operators perform either stateful (e.g., joins, aggregates) or stateless (e.g., map, filter) operations. By structuring computation as a dataflow graph, these systems efficiently manage concurrency, fault tolerance, and scalability.



(a) Stateful Entities (b) Cascade's Compiler

Figure 2: Compiler design high-level overview. The compiler transforms a source program composed of stateful entities (2a) into a DFG. The program statements refering to the `User` entity (marked in red in the source program) are mapped to dataflow nodes belonging to the `User` operator (anotated in red in the DFG). Similarly, program statements refering to the `Item` entity (marked in green) are mapped to dataflow nodes belonging to the `Item` operator (the green nodes). Cascade's compiler (2b) optimizes the IR before compiling it into specific datastream system code.

From Control Flow to Dataflow. While CFGs emphasize the execution order of basic blocks, dataflow graphs DFGs highlight the movement and transformation of data between dataflow operators. Basic blocks execute program logic when control reaches them; in contrast, dataflow operators react to incoming data, processing and potentially accumulating it

The code fragments within a basic block closely resemble the processing performed by a dataflow operator. The translation process begins by mapping code fragments from the CFGs basic blocks to dataflow operators. Next, a data analysis phase identifies which data elements are consumed and produced within the statements of each code fragment. This analysis determines how data flows between CFG operators and shares similarities with liveness analysis, where target variables and used values are extracted from each basic block. Finally, the last step involves identifying the data that must be accumulated by the dataflow operators, which in this context refers to modifications of entity attributes within the program state. A more detailed overview of the transformation process can be found in §4.

3 CASCADE'S INTERMEDIATE REPRESENTATION

Compilers for distributed systems can act as translators, transforming high-level stateful programs into efficient, distributed runtime environments. Such a transformation is typically facilitated by an intermediate representation (IR), which serves as an abstraction layer between the input program and the target system. Such an IR should ensure that the compiled program remains both expressive and optimizable for distributed execution.

IR Requirements. In order to define that an IR effectively supports distributed execution, it needs to fulfill some key requirements that we detail below. First, the IR must be expressive to allow for modeling a wide range of data-intensive applications. Second, the IR should enable optimization for distributed execution, where resource constraints and communication overhead are primary concerns. Third, an IR must explicitly capture stateful behavior, preserving the semantics of the original program even in a distributed environment. Fourth, since messaging can be expensive in distributed execution, the IR that we target should explicitly encode the dataflow between atomically connected code fragments, allowing the compiler to optimize inter-fragment communication. Lastly, the optimization framework should be modular and extensible, enabling future improvements that are independent to the IR structure.

To meet these requirements, we design our IR as a composition of two primary components: a set of Dataflow Operators that define computation and a collection of dataflow graphs that connect them. These components enable fine-grained control over execution and communication patterns, forming the foundation of an efficient distributed runtime. Figure 2 provides an overview of this compilation process.

In Cascade, the dataflow graph (DFG) defines the execution structure, defining how events propagate through the system. The DFG is directed, potentially cyclic, and encodes computation (nodes) and the flow of execution across Dataflow Operators (edges). The actual processing is carried out by dataflow operators, which maintain state across events and execute logic when triggered. The dataflow determines how events traverse through these dataflow operators, and ensures correct execution of stateful computations.

To explain in detail the construction of the dataflow graph, we use the MovieReview service from the DeathStarBench [26] Media Service workload as an example. This service comprises three stateless operators (Frontend, Text, UniqueId) and three stateful operators (ComposeReview, User, MovieId). The dataflow graph of ComposeReview is depicted in Figure 3. Now we will go through Figure 3 to describe the fundamental structure of the dataflow graph. A more detailed discussion on optimizing the dataflow from the Baseline (Figure 3a) to the Pipelined (Figure 3b) and Parallel (Figure 3c) versions is provided in §5.

Structure of the Dataflow Graph. At the core of Cascade's dataflow graph are OpNodes, the nodes representing the execution points. Each OpNode is linked to a specific Dataflow Operator and method. For example, the first blue node in the Baseline dataflow (Figure 3a) corresponds to the *UniqueId* operator and its *upload_unique* method. It is common for an operator to be visited multiple times within a single dataflow; for instance, in DeathStarBench, ComposeReview (the red node) is visited four times.

Certain OpNodes have specialized roles. The MovieId node (Figure 3) is an IfNode. IfNodes are conditional execution points that evaluate a predicate and route data accordingly. With two outgoing edges, an IfNode directs execution to either the **True** or **False** branch based on the predicate's outcome.

The Parallel Dataflow (Figure 3c) demonstrates how calls to external operators can be parallelized. In this transformation, we

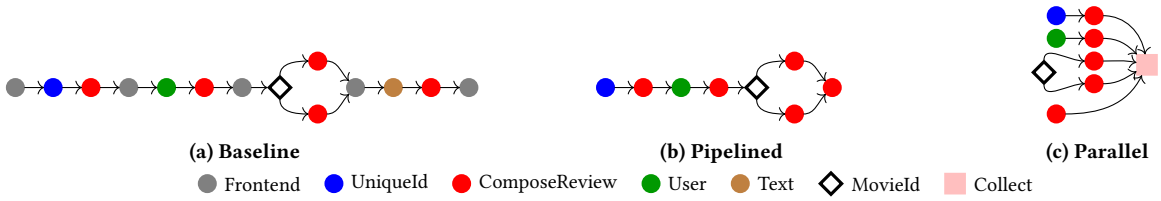


Figure 3: Dataflow graphs of the DeathStar benchmark, the baseline (a) and the two optimizaitons: pipelined (b) and parallel (c). The color of the nodes define to which operator they corespond.

introduce the Collect operator, a stateful component that collects results from multiple asynchronous inputs.

4 TRANSLATING SOURCE CODE TO IR

A source program composed of stateful entities is translated into the Cascades intermediate representation, which consists of two components: a list of dataflow operators and a dataflow graph, as described in §3. Dataflow operators are constructed from stateful entities, while the dataflow graph is derived from the program's control flow graph.

Creating dataflow operators from stateful entities is relatively straightforward, as their attributes can be directly extracted. Translating control flow into dataflow graphs, however, requires additional effort. While the basic blocks of a CFG closely resemble the code used to define the dataflow operators, a key difference lies in how they relate to stateful components: a basic block may reference multiple stateful entities, whereas each OpNode in the dataflow graph exclusively belongs to a single stateful operator.

To address this, we split basic blocks that reference multiple Stateful Entities. This involves inferring types and identifying when a class instance is a stateful entity. Additionally, we apply preprocessing steps to simplify code splitting.

From Stateful Entities to Dataflow Operators. Classes annotated with `@stateful` are recognized as stateful entities and are automatically registered for compilation. These classes typically contain both methods and attributes. The compiler translates methods into dataflow representations, while state updates to attributes define the data that needs to be accumulated by the dataflow operators.

Type Inference and Stateful Entity Identification. In the translation process, it is essential for cascade to understand which attributes include references to other stateful entities. To determine which entity type a variable belongs to the compiler must (1) infer its type and (2) check if that type corresponds to a stateful entity. Type inference requires code restrictions ensuring each variable's type is deducible. Cascade maintains a list of registered stateful entities; once the type is inferred, the check reduces to verifying the entity's presence in this list.

Pre-processing. To simplify the splitting of basic blocks, we transform the code into Three-Address Code (TAC) [6]. Next, we refine the statements to ensure that each statement contains only a single reference to a stateful entity invocation.

Cascade's CFG. The CFG used by Cascade is quite similar to a conventional CFG. However, we tailor the build of the CFG to the specific needs of our compiler. Cascade's purpose is to compile stateful entities into streaming dataflows; hence, we are only interested in the control flow concerning entities. When building a typical control flow graph, each `IfStatement` in the AST is mapped to an `IfBlock`. However, we choose only to build an `IfBlock` if the `body` or `orElse` branch of the `IfStatement` includes a reference to a stateful entity. The same applies to `ForLoop`.

Translating the CFG into a Dataflow. This pass splits the basic blocks into dataflow operators. At this stage, we can safely assume that each statement only references one stateful entity. The pass iterates through the basic block of the CFG, and any time the compiler encounters a stateful entity, it splits the basic block. The resulting splits form the nodes of the dataflow graph. The compiler visits the nodes once again and uses the type inference and stateful entity identification once more to assign each dataflow node to a dataflow operator.

5 OPTIMIZATIONS

This section outlines the optimization techniques applied to transform the Intermediate Representation (IR) into a more efficient form. These optimizations target Instruction-Level Parallelism (ILP) and Data-Level Parallelism (DLP) at the granularity of stateful entities. Since messaging incurs significant overhead in distributed execution, reducing unnecessary communication is a key objective. In the future, we plan to support Loop-level parallelism, code motion, liveness analysis, and possibly more optimizations.

Pipelining. The Baseline Dataflow (Figure 3a) illustrates how the `Frontend` operator orchestrates interactions between components, repeatedly returning control to itself after each call, introducing redundant calls and overhead. The pipelined version of the dataflow (Figure 3b) eliminates these calls. Additionally, the pipelined version removes the `Text` operator, as its `upload_text` method is only responsible for invoking `ComposeReview`. Therefore, we can directly call the `ComposeReview` operator:

```
class Text():
    @staticmethod
    def upload_text(review: ComposeReview, text: str):
        review.upload_text(text)
```

In essence, pipelining turns an orchestration into a choreography; this optimization is also illustrated in the example presented in Figure 1 of §1.

Parallelization. For the parallelization process, we can determine – using data dependency analysis – which calls to remote entities

can be called independently. If two method calls, do not use common state, they are considered to be data independent, and can be invoked in parallel. In our running example, it is possible to invoke the method invocations to `UniqueId`, `User`, `MovieId` and `Text` in parallel (Figure 3c).

Note that we assume to have access to the code of a complete application and side-effects to external systems do not exist [18]. We plan to work around this assumption in the future.

Loop-level parallelism. Analyzing loop-level parallelism at the granularity of stateful entities requires analyzing the independence of loop bodies invoking stateful entities. Consider, for example, a shopping cart with multiple items in the basket, and we wish to calculate the total cost. Each `Item` is an independent stateful entity containing an item price. Calculating the total could be done by iterating over the items, fetching the price, and accumulating the total. Since the loop body is independent, the compiler could optimize this to one broadcast for all the item prices and then collect the results.

Code Motion. In certain situations, code motion [21] Could reduce inter-entity communication and reduce message complexity. Consider the following example:

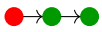
```
class User:
    def checkout_item(self):
        item = self.get_item_from_basket()
        price = item.price()
        balance = self.balance()
        stock = item.stock()
```

The red lines highlight program statements that involve access to the `User` entity. The green lines invoke the `Item` entity. In this case, the dataflow would move back and forth between the User Operator and Item Operator:

To minimize context switching between the `User` and `Item` entities, the statements can be reordered as shown below:

```
class User:
    def checkout_item(self):
        item = self.get_item_from_basket()
        balance = self.balance()
        price = item.price()
        stock = item.stock()
```

Resulting in a dataflow with a reduced number of nodes:



Liveness Analysis. In Cascade, while entity state is kept within their respective operators, *function* state travels with Events along the dataflow graph, and is responsible for saving the values of local variables. Liveness analysis of these local variables would minimize the size of the function state by only saving variables that will be used later. Additionally, this could also enable dead code elimination, potentially removing entire nodes from the dataflow graph.

6 FROM IR TO TARGET SYSTEMS

The IR is designed to be execution-target 'agnostic', allowing execution on any dataflow system such as Apache Flink [19], Apache Spark [38], Naiad [34] or Styx [37]. In this work we used PyFlink as the execution target, combining it with Kafka as a message queue to enable cyclic dataflow processing.

Method	Critical Path Length (CPL)	# Function Calls
Baseline	12	13
Pipelined	6	7
Parallel	2	8

Table 1: Critical path length and number of function calls for different optimizations.

In Flink, every `StatefulOperator` is compiled into a `KeyedProcessFunction`, enabling keyed state storage within the operator. The state corresponds to an instance of the Python class, allowing one to retrieve object properties and call methods on the underlying class instance. On the other hand, stateless operators, are transformed into `ProcessFunctions`. An ingress `ProcessFunction` is used to ingest and deserialize Events from the message queue and forward them to the correct operators.

Events directed toward `CollectNodes` are routed to a `FlinkCollectOperator`, implemented as a `KeyedProcessFunction`. This operator is keyed using a combination of the event IDs and the node IDs of the `CollectNode`. Events arriving at this node are buffered in the operator's state until all expected inputs have been received. Once all required Events are available, the `CollectNode` will yield the collection of Events.

The outputs of all operator datastreams are unioned before being sent to a Kafka sink. Depending on the event's progress through the dataflow, it is either re-ingested into the system for further processing or placed into a designated 'results' message queue, indicating that it has reached the end of its execution path.

Finally, the result of these operator datastreams are unioned before heading to a Kafka sink, where they will either be reingested into the system, or left in a "results" message queue if the event has reached the end of it's dataflow.

7 EVALUATION

Benchmark. To evaluate Cascade, we employ the composeReview workload from the DeathStar benchmark [26]. The workload consists of three stateless operators (`Frontend`, `Text`, `UniqueId`) and three stateful operators (`ComposeReview`, `User`, `MovieId`). We increase the request frequency from 200 to 1000 req/sec and measure the p50 and p99 round-trip latency of requests. A single request corresponds to an entire run-through of the dataflow, and therefore requires 7-13 individual function calls, as per Table 1. The `Text`, `UniqueId`, `User` and `MovieId` serve as intermediate operators as explained in §3.

Setup. Experiments are executed on a system with two 64-Core processors with 512 GB RAM, running 64-bit Ubuntu 22.04.5 and Flink version 1.20.1. Due to limitations in PyFlink's thread mode, we run 24 taskmanagers, each with only one task slot, in local Docker containers. To simulate more realistic distributed conditions, each taskmanager's resources were limited to 4 CPUs and 8GB RAM. They are coordinated by a single jobmanager. A single Kafka node is used for the message queue, using 32 partitions for each of the three required topics (`ComposeReview`, `User`, `MovieId`).

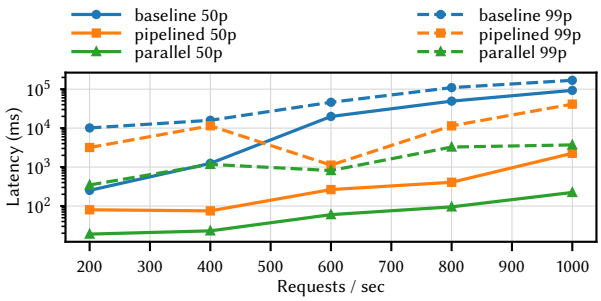


Figure 4: Evaluation of proposed optimizations compared baseline dataflow graph reporting 50p and 99p latency for different request frequencies ranging from 200 request/sec to 1000 req/second.

Results. The results are depicted in Figure 4. Firstly, the pipelined version benefits from halving the number of function calls (Table 1). In the p50 latency, the pipelining generates between 3x at 200 req/sec and 121x speedups at 800 req/sec. At 1000 req/sec, we notice that the speedup drops to 41x. We believe this break in the scaling behavior is a consequence of the backpressure on the baseline due to increased time spent on the Kafka queue. We argue that this behavior derives from performance limitations in PyFlink. The p99 latency experiences speedups of 1.4x to 41x. At 600 req/sec, we observe a considerable drop in the p99 latency, indicating a backpressure point in the pipelined version.

Next, we analyze the effects of Cascade's parallelization optimization. Introducing the additional Collect operator increases the total number of function calls by 1. Nonetheless, Cascade reduces the Critical Path Length (CPL) to 2. Thanks to the short critical path length, Cascade's parallelization optimization speeds up the execution by a further 3.2x to 10x. It proves to be the most effective at the highest loads of 1000 req/sec. Similarly, in the p99 latency, Cascade's parallelization achieves a speed up of up to 11x.

Finally, we discuss the combined benefits of Cascade's optimizations. In the p50 latency, Cascade speeds up the latency by 13x to 417x, and the p99 latency by up to 57x. Furthermore, the CPL is reduced from 12 to 2. In summary, our key takeaways are: *i*) minimizing unnecessary function calls in practice reduces overhead and improves execution efficiency, and *ii*) parallelizing computation effectively shortens the CPL, leading to 417x speedups at scale.

8 RELATED WORK

Cascade is inspired by the efforts of multiple research communities to remove the responsibility for correctness from programmers by providing recovery from failure. Specifically, we build upon concepts, such as static program analysis, compiler optimizations, dataflow machines, and fault-tolerant distributed system programming.

IRs and Optimizations in parallel dataflow systems. Optimizing dataflows in stream processing systems like Flink [19] and Spark [38] is challenging due to limited program context. Dataflow graphs often contain user-defined functions (UDFs), which are treated as black boxes, making it difficult for optimizations to be applied.

To address this, systems like Emma [5] and Mitos [27] leverage ANF-based intermediate representations (IRs) and embedded domain-specific languages (DSLs) in Scala. Similarly, DryadLINQ [25] introduces LINQ, a DSL that inherently treats data as parallel collections, enabling optimizations such as join order optimization and partial aggregation—tasks typically left to the programmer. The Mitos runtime extends these ideas to optimize iterative algorithms with techniques like loop pipelining and loop-invariant hoisting. However, these systems primarily focus on machine learning and big data analytics workloads. Another approach [30] applies static code analysis to inspect UDFs, extracting read-write sets to enable further optimizations. All these techniques operate on high-level algebraic DSLs, making their contributions orthogonal to Cascade. Unlike these systems, Cascade does not introduce a high-level algebra. Instead, its optimizations focus on improving the execution of stateful entities [23].

Compiler optimizations. Compilers were developed to automatically detect parallelism and leverage multi-processor computer architectures [35]. Combinations of data dependency testing [8, 20, 28] and control-flow analysis [3, 7] were applied, to check for *instruction-level* and/or *data-level* parallelism. Instruction-level parallelism analyses a sequence of operations to extract hidden parallelism [29], while data-level parallelism can often be leveraged when processing large volumes of data. Multiple techniques that belong to these categories inspired Cascade, such as loop unrolling, Loop Interchanging, Fission by Name and Loop Fusion, Loop Collapsing.

Programming models. As distributed systems became more prevalent, new programming languages emerged to simplify their development [12]. Languages such as Distributed ML [32], Erlang [11] and Smalltalk [24] were designed to provide abstractions for fault tolerance, concurrency, and message passing, making it easier for programmers to build reliable and efficient distributed applications.

In addition to new programming languages, modern reincarnations of the Actor programming model [16, 17, 43] emerged as an alternative approach to modeling distributed system behavior. Altough the Cascades programming model is inspired by Orleans Virtual Actors, we differ from them by applying program analysis and optimizations at granularity of the Entity. Orleans has asynchronous constructs and the capabilities to create promises, however it is up to the programmer to do so in an efficient way. We draw inspiration from Liskovs Promises [33], who developed a method for pipelining remote procedure calls into something they coined *call streams*.

Compiling imperative programs to dataflow. In the late 1980s and early 1990s, researchers explored executing imperative programs on dataflow machines to enable scale-up multi-processor computing [13, 42]. Their approach employed a fine-grained method, transforming each operation into a dataflow operator. In contrast, Cascade translates code at a coarser granularity, mapping one operator per stateful entity and focusing on scaling out distributed computing.

9 CONCLUSION AND FUTURE WORK

In this paper, we present Cascade, a compiler pipeline that transforms stateful imperative code into dataflows. Specifically, we demonstrate how static code analysis and optimization at the level of an intermediate representation (IR) can enhance performance and simplify cloud application development. Evaluating Cascade on the DeathStar benchmark Media Service indicates an average p50 and p99 latency speedup of 267x and 36x, respectively.

Since our research is in progress, multiple challenges remain open. A main challenge in optimizing dataflow graph execution is handling side effects regarding state. In order to handle side effects, a more detailed analysis is needed to determine when functions exhibit indirect dependencies (e.g., modifying the same data structure) that can impose execution constraints. To address this challenge, we plan to extend Cascade with side-effect analysis [18]. Additionally, advanced data dependency analysis will enable further optimizations, including code motion to reduce inter-entity communication and liveness analysis to minimize the size of the function state that travels through the dataflow.

Furthermore, we aim to apply optimizations inspired by compiler design, which also fit the distributed nature of dataflow runtimes. The goal is to reduce redundant state accesses, minimize entity calls, and increase parallelism. For instance, dynamic analysis could enable branch prediction and speculative execution, preemptively fetching state to enhance performance.

In addition, our goal is to extend our evaluation with more workloads and targeted systems. We aim to include additional target systems to highlight the flexibility of our IR, as well as, support workloads that include entity queries (e.g., the Hotel Service in the DeathStar benchmark). For the latter, we will introduce a `QueryNode`, enabling stateful entity queries. Additionally, leveraging program information in the IR (e.g., distinguishing read-only nodes) could enable more optimizations.

Lastly, once we establish a stable IR design, we will shift our focus to dynamic system deployments. Currently, we assume round-robin state distribution. In future research, we aim to investigate state migration and state placement strategies, leveraging the stateful dataflow graph to optimize state and execution co-location based on a cost model.

REFERENCES

[1] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *the VLDB Journal* 12 (2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] Sethi Aho, Ravi Sethi, and D Jeffrey. 1986. Ullman, Compilers: Principles, Techniques, and Tools.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. 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 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[5] Alexander Alexandrov, Georgi Krastev, and Volker Markl. 2019. Representations and optimizations for embedded parallel dataflow languages. *ACM Transactions on Database Systems* 44 (1 2019). Issue 1. https://doi.org/10.1145/3281629

[6] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools*. pearson Education.

[7] Frances E. Allen. 1970. Control flow analysis. *SIGPLAN Not.* 5, 7 (July 1970), 1–19. https://doi.org/10.1145/390013.808479

[8] John R Allen and Ken Kennedy. 1982. PFC: A program to convert Fortran to parallel form. (1982).

[9] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 223–236. https://doi.org/10.1145/1755913.1755937

[10] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[11] Joe Armstrong. 2013. *Programming Erlang: software for a concurrent world*. The Pragmatic Bookshelf.

[12] Henri E Bal, Jennifer G Steiner, and Andrew S Tanenbaum. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys (CSUR)* 21, 3 (1989), 261–322.

[13] Micah Beck, Richard Johnson, and Keshav Pingali. 1991. From control flow to dataflow. *J. Parallel and Distrib. Comput.* 12, 2 (1991), 118–129. https://doi.org/10.1016/0743-7315(91)90016-3

[14] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR*.

[15] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485510

[16] Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. 2010. Orleans: A framework for cloud computing. *URL: https://www. microsoft. com/en-us/research/wp-content/uploads/2010/11/pldi-11-submission-public.pdf* (2010).

[17] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

[18] David Callahan and Ken Kennedy. 1987. Analysis of interprocedural side effects in a parallel programming environment. In *International Conference on Supercomputing*. Springer, 138–171.

[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 36, 4 (2015).

[20] Shyh-Ching Chen et al. 1979. Time and parallel processor bounds for Fortran-like loops. *IEEE Trans. Comput.* 100, 9 (1979), 660–670.

[21] Cliff Click. 1995. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. 246–257.

[22] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. *ACM Sigplan Notices* 30, 3 (1995), 35–49. https://doi.org/10.1145/202530.202534

[23] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems* (Virtual Event, Italy) *(DEBS '21)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3465480.3466920

[24] L Peter Deutsch and Allan M Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 297–302.

[25] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. 2009. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR* 8 (2009).

[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA)

(ASPLOS '19). Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[27] Gabor E. Gevay, Tilmann Rabl, Sebastian Bres, Lorand Madai-Tahy, Jorge Arnulfo Quiane-Ruiz, and Volker Markl. 2021. Efficient control flow in dataflow systems: When ease-of-use meets high performance. In *Proceedings - International Conference on Data Engineering*, Vol. 2021-April. IEEE Computer Society, 1428–1439. https://doi.org/10.1109/ICDE51399.2021.00127

[28] W Ludwell Harrison III. 1986. *Compiling lisp for evaluation on a tightly coupled multiprocessor.* Technical Report. Illinois Univ., Urbana (USA). Center for Supercomputing Research and Development.

[29] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach.* Elsevier.

[30] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the black boxes in data flow optimization. *Proc. VLDB Endow.* 5, 11 (July 2012), 1256–1267. https://doi.org/10.14778/2350229.2350244

[31] Richard M. Karp and Rayamond E. Miller. 1966. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411. https://doi.org/10.1137/0114108 arXiv:https://doi.org/10.1137/0114108

[32] Clifford Dale Krumvieda. 1993. *Distributed ML: abstractions for efficient and fault-tolerant programming.* Technical Report. Cornell University.

[33] B. Liskov and L. Shrira. 1988. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.* 23, 7 (June 1988), 260–267. https://doi.org/10.1145/960116.54016

[34] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[35] David A. Padua and Michael J. Wolfe. 1986. Advanced compiler optimizations for supercomputers. *Commun. ACM* 29, 12 (Dec. 1986), 1184–1201. https://doi.org/10.1145/7902.7904

[36] Kyriakos Psarakis, George Christodoulou, Marios Fragkoulis, and Asterios Katsifodimos. 2025. Transactional Cloud Applications Go with the (Data) Flow. In *15th Annual Conference on Innovative Data Systems Research (CIDR'25)*. VLDB Endowment.

[37] Kyriakos Psarakis, George Christodoulou, George Siachamis, Marios Fragkoulis, and Asterios Katsifodimos. 2025. Styx: Transactional Stateful Functions on Streaming Dataflows. In *ACM SIGMOD 2025*.

[38] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1, 3 (2016), 145–164.

[39] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[40] James Stanier and Des Watson. 2013. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.* 45, 3, Article 26 (July 2013), 27 pages. https://doi.org/10.1145/2480741.2480743

[41] Linda Torczon and Keith Cooper. 2007. *Engineering a compiler.* Morgan Kaufmann Publishers Inc.

[42] Arthur Hugo Veen. 1985. The Misconstrued Semicolon: reconciling imperative languages and dataflow machines. (1985).

[43] Derek Wyatt. 2013. *Akka concurrency.* Artima Incorporation.

[44] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. 2006. Hilda: A high-level language for data-drivenweb applications. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 32–32. https://doi.org/10.1109/ICDE.2006.75

[45] Kenneth Zadeck. 2009. The development of static single assignment form. In *Static Single-Assignment Form Seminar*.

[46] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709