Automated Verification of Idempotence for Stateful Serverless Applications

Haoran Ding[1], Zhaoguo Wang[1], Zhuohao Shen[1], Rong Chen[1,2], and Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Shanghai AI Laboratory

Abstract

Serverless computing has become a popular cloud computing paradigm. By default, when a serverless function fails, the serverless platform re-executes the function to tolerate the failure. However, such a retry-based approach requires functions to be idempotent, which means that functions should expose the same behavior regardless of retries. This requirement is challenging for developers, especially when functions are stateful. Failures may cause functions to repeatedly read and update shared states, potentially corrupting data consistency.

This paper presents Flux, the first toolkit that automatically verifies the idempotence of serverless applications. It proposes a new correctness definition, *idempotence consistency*, which stipulates that a serverless function's retry is transparent to users. To verify idempotence consistency, Flux defines a novel property, *idempotence simulation*, which decomposes the proof for a concurrent serverless application into the reasoning of individual functions. Furthermore, Flux extends existing verification techniques to realize automated reasoning, enabling Flux to identify idempotence-violating operations and fix them with existing log-based methods.

We demonstrate the efficacy of Flux with 27 representative serverless applications. Flux has successfully identified previously unknown issues in 12 applications. Developers have confirmed 8 issues. Compared to state-of-the-art systems (namely Beldi and Boki) that log every operation, Flux achieves up to 6× lower latency and 10× higher peak throughput, as it logs only the identified idempotence-violating ones.

1 Introduction

A serverless application typically comprises a collection of functions, which may be stateful. For example, they may communicate with each other through a shared database. Major serverless platforms generally support the stateful model, such as AWS [17], Microsoft [59], and Google [36]. Platforms generally employ a retry-based fault tolerance mechanism for stateful applications — they automatically retry a function in case of an unexpected error [18, 35, 57].

However, this mechanism mandates developers to write *idempotent* applications that produce consistent results irrespective of the number of retries. In a sequential system that invokes functions sequentially, developers can reason about each function independently. However, a concurrent system can invoke functions simultaneously. Therefore, developers must consider all possible interleavings of concurrent functions, making it challenging to write idempotent applications.

This paper presents Flux, the first toolkit that automatically verifies the idempotence of concurrent serverless applications. Building such a toolkit posed several challenges. First, a formal idempotence definition for concurrent systems is desired but currently missing. Second, automated verification requires examining all possible interleavings of concurrent serverless functions with arbitrary failures, which is prohibitively expensive. Third, for non-idempotent applications, the toolkit should accurately identify the code that corrupts idempotence, enabling developers to fix the issues.

To overcome the first challenge, we propose a novel idempotence definition for concurrent systems — *idempotence consistency*. A serverless application is idempotence-consistent if, for any observable behavior of an execution with retries, there exists another execution without retries that can produce the same behavior. Achieving idempotence consistency makes clients unaware of retries during execution (Section 3). Unlike alternative idempotence conditions, such as exactly-once execution [45, 73], idempotence consistency offers greater flexibility. An idempotence-consistent application does not necessarily ensure exactly-once execution of *all* database operations.

To tackle the second challenge, we propose *idempotence simulation* to realize compositional proof, which enables proving the idempotence consistency of an application by verifying each function individually. For each function, Flux verifies that every possible execution with retries has a corresponding retry-free execution that can simulate it (Section 4). Existing work [66] can realize automated verification in a sequential system by comparing the execution results when retries happen with the results without retries. However, in a concurrent system, verification requires modeling the con-

current environment to account for the side effects of running multiple functions concurrently. Unfortunately, existing modeling approaches are not fully automated as they ask developers for hints, such as invariants [40, 60] and rely conditions [47, 53]. In contrast, Flux automates the generation of rely conditions and other hints. Additionally, Flux proposes *failure reduction* to avoid enumerating all failure cases.

To help fix idempotence issues, Flux can identify idempotence-violating operations whose re-execution corrupts idempotence consistency. Developers can use logs to ensure exactly-once execution semantics for these operations rather than all operations via existing mechanisms [45, 73].

We evaluate Flux on 27 representative serverless applications with 79 functions. These applications are from various sources, such as the AWS serverless application repository [3], a GitHub repository (10.9k stars) [9], popular serverless benchmarks [8, 10], and applications commonly used in papers about serverless computing [5, 6]. Flux successfully identifies previously unknown idempotence issues in 12 applications. Compared to state-of-the-art systems (namely Beldi [73] and Boki [45]) that log all operations, Flux achieves up to 6× lower latency and 10× higher peak throughput, as it logs only identified idempotence-violating operations.

Nevertheless, Flux still has several limitations. First, although we design it for stateful serverless functions, its assumption that states only include data in NoSQL databases restricts its applicability to storage systems of other types. We need to model the semantics of other storage systems carefully. Second, our verification method is sound but incomplete since Flux cannot handle certain serverless functions, such as functions having certain types of unbounded loops. Last, Flux currently supports only Java applications since we build Flux based on a symbolic execution engine for Java applications [61]. Despite these limitations, we believe that Flux takes an important step towards enabling verification for idempotence consistency of serverless functions.

2 Motivation and Our Approach

Why Need Idempotence? The concept of idempotence is crucial for applications that rely on retry-based methods to tolerate failures. Without idempotence, re-executing failed computations may result in unexpected side effects, causing severe correctness issues [42, 48, 64, 67]. With the emergence of serverless computing, idempotence has become a significant requirement for serverless applications. However, this requirement poses a challenge when serverless functions run concurrently and are stateful, placing a substantial burden on using serverless platforms [44, 69, 73].

We use an example in Figure 1 to illustrate why re-executing a non-idempotent application can cause issues. This is an example derived from a real-world e-commerce web application, Spree [68]. The *payment* function atomically checks the customer's balance and deducts the price with a discount rate using a conditional update API (line 4). This

```
1  void payment(productId, userId, price) {
2    discount := get("Discount", productId);
3    total := price * discount;
4    success := cond_update("Balance", userId,
5                  inc(-total), gte(total));
6    receiptId := generateId(userId, productId,
7                   localTime());
8    if(success)
9      put("Receipt", receiptId, total);
10 }
11
12 void adaptDiscount(productId, percent) {
13   if(!isValid(percent))
14     return;
15   discount := 1.0 - percent/100.0;
16   put("Discount", productId, discount);
17 }
```

Figure 1: A simplified e-commerce serverless application with two functions. When *balance* in the database is greater than or equal to *total* (*gte(total)* is true), *cond_update* (line 4) decreases *balance* by *total* (*inc(-total)*) and returns true. Otherwise, it returns false. The *generateId* function (line 6) returns a receipt identifier. The *isValid* function (line 13) returns true iff *percent* is between 0 and 100.

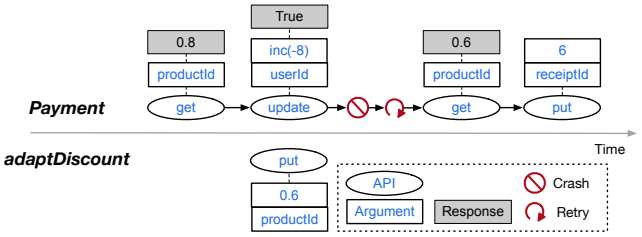
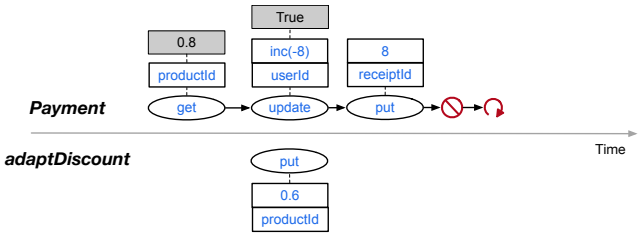


Figure 2: The concurrent execution of *payment* and *adaptDiscount* in Figure 1.

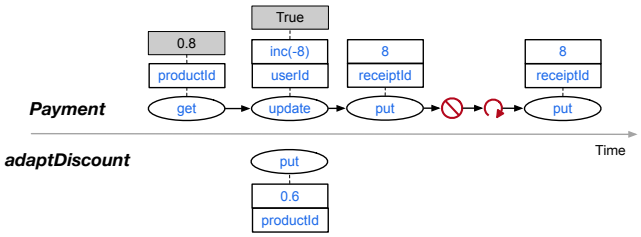
method needs to create a receipt accordingly (line 9). Meanwhile, the *adaptDiscount* function changes the discount for a specific product, which the seller typically invokes.

Suppose a failure occurs after *payment* deducts the price from the customer's balance at line 4 but before it creates a new receipt at line 9. Consider the sequential execution of *payment* without concurrency. If the platform re-executes this function after the failure, the function will deduct the price twice from the customer's balance. One possible solution to ensure idempotence is to log the conditional update operation. Additionally, it is necessary to log the execution of *generateId*, as *localTime* returns different values on retry. These logs can ensure that the function will not re-execute *update* (line 4) and *generateId* (line 6) on retry, which will always return the same value as the first execution. The solution is enough to provide idempotence under the sequential scenario.

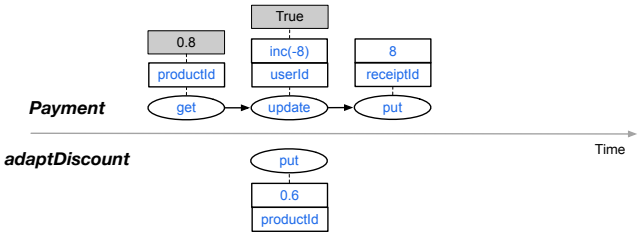
However, the above solution does not work when *payment* runs concurrently with *adaptDiscount*. For example, suppose *adaptDiscount* changes the discount after *payment* fails at line 4 but before its re-execution. Then, *payment* would have deducted the price with the old discount but created a new receipt with the new discount, which poses an inconsistency between the customer's balance and the corresponding receipt. Figure 2 shows the specific interleaving. Flux aims to automatically find the correct logging strategy under both



(a) The execution with a retry when logging all operations of *payment*.



(b) The execution with a retry when logging all operations of *payment* except for the *put*.



(c) The normal execution without retries and logging operations.

Figure 3: Three different concurrent executions of the functions in Figure 1. The legend is the same as that in Figure 2.

sequential and concurrent scenarios via verification.

Idempotence Condition. Some recent efforts have focused on the retry-based fault tolerance mechanism for serverless applications [44, 45, 69, 73]. Although they optimize runtimes or libraries of serverless computing, they overlook the definition of idempotence. For example, Beldi [73] and Boki [45], which contribute novel distributed logging mechanisms, equate idempotence with executing each database operation exactly once. They achieve this by logging every operation to ensure that functions execute each operation only once. However, repeating some operations does not compromise idempotence. For instance, as shown in Figure 3b, logging all operations except for *put* on *receipt* can still ensure idempotence, as *payment* will always write the same value into the receipt on retry as it did in its first execution. However, as illustrated in Figure 3a, Beldi and Boki need to log every operation, which is over-restricted and incurs unnecessary performance costs. This logging strategy misses the opportunity to maximize performance while ensuring idempotence.

When defining idempotence for serverless applications, we should consider what kind of execution with retries is acceptable. The intuitive requirement is that clients should be unaware of retries. This requirement enables us to define acceptable execution with retries in terms of normal execution without retries. For instance, the executions with a retry in Figure 3b and Figure 3a are both acceptable because they are equivalent to the normal execution in Figure 3c. However, the execution in Figure 2 is unacceptable because we cannot find an equivalent normal execution for it. The normal concurrent execution of *payment* with *adaptDiscount* will never deduct "8" from the balance but record "6" in the receipt. Therefore, determining whether an application is idempotent requires checking whether any possible execution with retries is acceptable.

Verification of Idempotence. Several frameworks for verifying storage systems also prove the idempotence of recovery functions [23, 24, 27, 28, 66]. Specifically, the resulting state of the recovery should be consistent even if the system fails during recovery and retries the recovery function many times. The work based on Crash Hoare Logic [23, 24, 27, 28] verifies idempotence by proving that the crash condition of the recovery function always implies its precondition. Developers need to specify both pre and crash conditions manually. The push-button verification approach [66] frees developers from such a proof burden by automatically verifying recovery functions with SMT solvers. However, all such methods assume that the recovery procedure is sequential. Verifying the idempotence of concurrent functions is still missing.

To prove the idempotence of concurrent functions, we use compositional proof techniques. The fundamental idea is to break down the verification of an application's idempotence into verifying each function individually. However, the main challenge is defining the property that needs verification for each function, which can facilitate compositional proof. Besides, modeling the behavior of other concurrent function instances also poses a challenge. Existing methods typically use invariants [60] or rely conditions [47, 53] to model the concurrent environment. Invariants describe the properties of the system state that persist during concurrent execution, while rely conditions depict how other concurrent functions can change the system state. Unfortunately, developers must explicitly specify all of them. We need to infer invariants or rely conditions automatically for automated verification.

Our Approach. To define idempotence, we propose a new consistency model called *idempotence consistency* (Section 3). An execution with retries is acceptable if there exists another normal execution without retries that can exhibit the same observable behavior (e.g., Figure 3b). An application is idempotence-consistent if all possible executions with retries are acceptable. To verify idempotence consistency, we propose idempotence simulation (Section 4), which extends traditional forward simulation [55] and enables compositional proof for idempotence consistency. Specifically, the verification process tries to find a mapping from each step during the execution with retries to n steps during the execution without retries such that the single step and the n steps exhibit the same observable behavior.

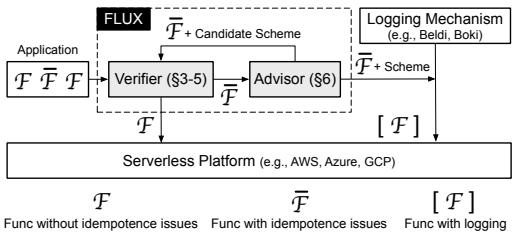


Figure 4: The architecture of Flux.

```
1 bool retry := random();
2 if(retry)
3 {
4   reset_local_state();
5   goto BEGIN;
6 }
```

Figure 5: The pseudocode simulating random failures and retries of a function f. BEGIN is a label at the beginning of f.

Figure 4 shows the components of Flux. First, developers provide the source code of a serverless application for Flux. Then, Flux checks each function individually to reason about idempotence simulation (Section 5). If all functions pass verification, the application is idempotence-consistent. If not, advisor identifies operations that corrupt idempotence consistency based on the results of the verifier (Section 6). Developers can use existing logging mechanisms to ensure exactly-once semantics of such operations. Compared with logging all operations pessimistically, Flux guarantees idempotence while reducing unnecessary protection overhead.

3 Idempotence Consistency

Idempotence consistency requires that each execution with retries should have the same observable behavior as another execution without retries. To formally define idempotence consistency, Flux uses an automaton to model the concurrent execution of functions. An automaton includes system states and a set of steps. Each step (S, e, S') represents a state transition from state S to S', which triggers an event e observable to clients (e.g., a function invocation). Note that some steps may not produce events because they are not observable to clients.

System State. The system state consists of the shared state and the local state of each function instance. In scenarios where functions use a NoSQL database, the shared state D constitutes a collection of key-value pairs stored persistently in databases. Given an instance executing a function, its local state includes the invocation arguments, the local variables, the return value, and the next program statement to execute. The start system state only contains the shared state because the platform has not invoked any functions at the beginning.

Event. An automaton may produce an event during each state transition, which is observable to clients. In the context of serverless functions, Flux considers three types of events: function invocation events, function response events, and third-party service events. When creating a new instance $f_{id}(args)$ to run the serverless function f, the automaton produces an invocation event $(f_{id}, inv(args))$. When the instance f_{id} finishes its execution and successfully responds to clients with the value v, it produces a response event $(f_{id}, resp(), v)$. When the serverless function requests a third-party service s, it produces a third-party service event $(id_s, args, ret)$. The $args$ and ret represent parameters and return values, respectively. An automaton produces such events when a service has side effects, which developers must explicitly specify.

Client-Observable Behavior. The client-observable behavior of an execution includes all events generated during the execution and the final shared state observable by clients. The events include function invocation, response, and third-party service events. We use $\langle H, D \rangle$ to denote the client-observable behavior, where H represents the event sequence generated by the automaton throughout the execution, and D is the shared state reached after the execution.

Given a function set F, to model the execution of functions in F under failure, we rely on the following failure model and star operator.

Failure Model. Flux assumes that failure can occur at any time during the execution of an instance. The failure of individual instances does not affect the persistent shared state or individual local states of other instances. Furthermore, when the platform retries an instance, it retains the same identifier and arguments, generating no invocation events.

Star Operator. Given a function f, f^* denotes a function synthesized by inserting a code fragment after every statement. When random failures and retries occur during a function's execution, the platform will re-initialize the local state and re-execute the function from the first statement (without retry events). The code fragment (Figure 5) simulates random failures by resetting the local state and simulates retries by jumping to the beginning of f. F^* is a function set synthesized by applying the star operator to each function in F.

Based on the above concepts, Flux is able to define idempotence consistency as the relationship between two automata. It means that F allows all possible client-observable behaviors for the automaton of F^*.

Definition 1 (Idempotence Consistency) For any start system state S and any step sequence of the automaton executing F^* from S, if the step sequence results in the client-observable behavior $\langle H, D \rangle$, then there always exists another step sequence of the automaton executing F from S such that it also results in $\langle H, D \rangle$.

4 Proof Strategy

Using compositional proof techniques [52–54, 76], Flux automatically verifies the idempotence consistency of serverless applications. The fundamental idea is to simplify the proof of a concurrent program by reasoning about each of its components separately. Several existing approaches utilize compositional proof to verify the correctness of concurrent programs, such as RGSim [53], AtomFS [76], and Armada [54].

However, the primary difference between Flux and existing approaches is that existing approaches require human experts to aid the proof, such as manually specifying the correctness definition or modeling program behavior under concurrency. Flux, on the other hand, performs entirely automated verification without human intervention.

Preliminary. Flux adopts compositional proof techniques, verifying a concurrent program by checking each component individually rather than enumerating all possible interleavings. For instance, when verifying a concurrent stack implementation [52], programmers only need to separately consider the correctness of *push* and *pop* functions under concurrency. To perform compositional proof, programmers first need to manually specify each component's expected behavior (e.g., specifications for *push* and *pop* functions). Second, programmers should carefully craft the pre- and post-conditions of each statement, which are propositions describing the system state before and after executing the statement. The verification goal is that when the start system state satisfies the pre-condition of the first statement, the system state after executing the component must satisfy the post-condition of the last statement under concurrency. Note that other concurrent threads can simultaneously modify the system state. Therefore, to consider all possible execution results, pre- and post-conditions must cover all possible system states before and after executing a statement. Developers need to prove that the pre- and post-conditions are stable under concurrency, which means they always hold irrespective of how other concurrent functions simultaneously modify the system state. Finally, to verify the stability of pre- and post-conditions, programmers must manually define a rely condition R, which describes the state transition made by other concurrent threads. R is a relation of system states. Each (S, S') in R indicates that other concurrent threads might change the current state S to S'. In the example of the stack, the rely condition specifies the impact of concurrent *push* and *pop* operations on the global linked list that represents the stack. We can define it as $\{(\ell, \ell') \mid ((\exists v. \ell' = PUSH(\ell, v)) \vee (\ell' = POP(\ell)))\}$. ℓ represents the state of the linked list, while ℓ' is the new state after applying the operations of $PUSH(\ell, v)$ or $POP(\ell)$.

4.1 Idempotence Simulation

Instead of manually crafting the specification, Flux introduces a new correctness definition — *idempotence simulation*. A serverless function f satisfies idempotence simulation if there exists a forward simulation [55] between f^* and f under the same rely condition R. The forward simulation means that from the same start shared state with the same invocation argument, each step of executing f^* has zero or multiple corresponding steps of executing f that can simulate it. If a step s of f^* changes the shared state D to D' and produces an event e, a step sequence $s_1 \dots s_i$ of f can simulate it if and only if carrying out these steps sequentially from D also reaches D' and produces e. Note that idempotence simulation does not only consider the shared state reached by executing the current instance but also D reached by executing other concurrent instances according to R. Flux differs from previous verification frameworks [23, 24, 27, 28, 66] that focus on the idempotence of sequential functions by considering intermediate states. This difference is significant because, in a concurrent setting, these intermediate states may be externally observable.

Given a function set F, Flux decomposes the proof of idempotence consistency into reasoning about the idempotence simulation of every function f in F based on the following theorem. $f^* \sqsubseteq_R f$ means that for any start shared state and invocation argument, there exists a forward simulation between f^* and f under the same rely condition R.

Theorem 1 Given a function set F, if each function f in F satisfies idempotence simulation, then F satisfies idempotence consistency, denoted as the predicate idem(F).

$$(\forall f \in F. \exists R. f^* \sqsubseteq_R f) \rightarrow idem(F).$$

Appendix A presents the formal proof of Theorem 1. We only illustrate its intuition as follows. Existing work [53] based on rely conditions has proposed methods to prove that the specification can exhibit all possible observable behaviors of the implementation. When proving idempotence consistency, we observe that we can treat F^* as the implementation and F as its specification. Then we can utilize the compositional proof technique in the existing work to verify idempotence consistency. An important fact is that the steps of implementation consist of the steps from each component, while the specification of the implementation consists of the specification of each component. Therefore, the existing work first proves that for each component and its specification, each step of the component has zero or multiple steps of the specification that can simulate it under the rely condition R. Then the verifier can compose the proof for each component to imply that each step of the implementation always exhibits the observable behavior allowed by its specification. In the scenario of idempotence consistency, we treat each serverless function f^* as the component. Then we can treat f as the specification for f^*. Based on the observation, Flux defines idempotence simulation as the forward simulation between f^* and f, which can imply idempotence consistency.

4.2 Automated Concurrency Reasoning

To verify the program with compositional proof, existing approaches [52–54, 76] require programmers to manually define the pre- and post-condition of each statement, as well as the rely condition for concurrent state transitions. The cause of the need for manual effort is that different definitions of correctness usually require different pre-, post-, and rely conditions. Programmers need to deeply understand the definition of correctness to find and specify the appropriate conditions.

However, idempotence consistency presents a unified correctness definition for stateful serverless functions, establish-

ing an opportunity to generate pre-, post- and rely conditions automatically. These conditions capture the potential concurrent accesses to the shared database state and describe the state transition on each access. Flux accomplishes this with symbolic execution. To reduce the complexity of analysis, Flux models the semantics of each API as a sequence of atomic read operations or atomic write operations on a set of key-value pairs, as most serverless platforms use NoSQL databases with key-value interfaces [16, 34, 58]. Then, it can symbolically execute all functions and check the parameters of issued database operations to analyze the data in the shared state that functions can access. Section 5 will depict more details.

Flux identifies three types of rely conditions.

- Read-only: all concurrent accesses to a specific key-value pair are read operations. Flux formalizes this type of rely condition as $(D[k] = v,\ D[k] = v)$, which means before and after the access, the value (v) indexed by k in the database keeps unchanged.
- Arbitrary update: functions could update the data in the database to arbitrary values. Flux formalizes this type of rely condition as $(\exists v_1.D[k] = v_1,\ \exists v_2.D[k] = v_2)$.
- Constant update: functions will update the specific key-value pair to only a constant value. Flux formalizes this type of rely condition as $(\exists v_1.D[k] = v_1,\ D[k] = c)$, where c is the constant value.

Flux constructs pre- and post-conditions in the Floyd-Hoare style ("$\{P\}C\{Q\}$"). C is the next program statement to execute. P is the pre-condition before executing C, while Q is the post-condition after the execution. If C is the first statement of the function, then P is *true*. Flux adopts the following rules to automatically generate the pre- and post-conditions according to the semantics of C and different rely conditions:

- $\{P\}put(k,c)\{P \wedge (\mathrm{D[k]=c})\}$: if the rely condition specifies that all concurrent updates are constant updates with a constant value of c, then $D[k]$ will always be c after executing $put(k,c)$.
- $\{P\}v := get(k)\{P \wedge (\mathrm{D[k]=v})\}$: if the rely condition specifies that all concurrent accesses to k are read operations, then the value of $D[k]$ should be exactly v which is the return value of the *get* operation in C.
- $\{P\}if(P_1(v))\{P \wedge P_1(D[k])\}$: suppose C is a branch statement based on $P_1(v)$, and v is the value read from the database, indexed by k. The post-condition is $P \wedge P_1(D[k])$ if $D[k]$ satisfies one of the following requirements according to rely condition: 1) $D[k]$ is read-only; 2) functions can update $D[k]$ to a constant value c such that $P_1(c)$ is true.
- $\{P\}C\{P\}$: in the other cases, the post-condition is the same as the pre-condition, which is stable.

4.3 Unbounded Loop

Another challenge in automated verification involves dealing with functions that contain unbounded loops. A loop is unbounded when its maximum number of iterations is not constant. Existing approaches require that programmers manually specify loop invariants to handle unbounded loops. However, previous works have shown that finding a proper loop invariant is challenging [19, 33]. Flux reasons about unbounded loops without requiring loop invariants. The following paragraphs provide the details in two cases:

Case 1. The operations in the unbounded loop do not modify the shared state in the database. For this case, Flux treats the entire unbounded loop as an uninterpreted function [12], which is a symbolic function and may return arbitrary values. Specifically, Flux derives a new function g from the original function f by replacing the unbounded loop with an uninterpreted function. Then, Flux directly reasons the idempotence simulation for g instead of f.

Case 2. The operations in the unbounded loop may modify the shared state. Flux addresses this type of unbounded loop with Theorem 2, which requires that the parameters of the write operations in such unbounded loops must remain the same between normal execution and retry. For convenience, Flux represents a function with an unbounded loop as $\{C_1;L;C_2\}$, where L is the unbounded loop, C_1 denotes all code before the loop, and C_2 denotes all code after the loop. B_L denotes the loop body of L. We present the theorem as follows. Appendix C provides the formal proof and an example of applying the theorem.

Theorem 2 Given a function f with the unbounded loop in case 2, f satisfies idempotence simulation if the number of iterations of the loop L remains unchanged on retry, and C_1, C_2 and B_L can satisfy the following requirements: 1) They all satisfy idempotence simulation; 2) Their inputs do not change on retry; 3) They will not affect the shared state on retry once the function has successfully executed them.

4.4 Failure Reduction

The platform may re-execute a function an arbitrary number of times. It is impossible to verify idempotence simulation if we enumerate all possible failure cases, which yields infinite possible executions. Instead, we prove that it is sufficient to verify a function only by examining the executions satisfying two conditions, as stated in the following theorem.

Theorem 3 For any function set F, if each function $f \in F$ satisfies idempotence simulation under the following two conditions: 1) failure happens only after statements modifying the shared state, and 2) failure occurs at most once, then each function $f \in F$ also satisfies idempotence simulation under arbitrary failure and retries.

This result (i.e., failure reduction) mitigates the challenge of proving idempotence simulation with infinite failure. It

```
bool retry := random();
if(retry && hasretried < LIMIT)
{
  reset_local_state();
  hasretried++;
  goto BEGIN;
}
```

Figure 6: The pseudocode simulating random failures and retries of f^n.

transforms the problem of examining executions with infinite failure into the problem of reasoning about executions with finite failure. Moreover, it maintains the soundness of the verification, signifying that Flux does not overlook any idempotence issues.

The intuition behind the first condition is that a statement that does not modify the shared state lacks side effects if a failure occurs after it. Because the failure effectively renders the result of the statement invisible to clients and the following code. Therefore, when the failure occurs, it appears as if the function instance never executed the statement.

The second condition is correct and ensures soundness. We formalize its correctness based on the following concepts. Given a function f, f^n denotes the function whose number of re-execution is not more than n times ($n \geq 0$). We can construct f^n by inserting the code fragment in Figure 6 after every statement of f. The global variable *hasretried* is initially zero, which indicates the number of retries that have occurred. We can simulate n retries for f^n by setting the constant *LIMIT* to n. The correctness of the second condition follows from the following theorem. Appendix B presents the formal proof.

Theorem 4 Given a function f in F, if the execution of f can simulate f^1 under concurrency, then for any $n \geq 1$, the execution of f can simulate f^n under concurrency.

$$(\exists R.f^1 \sqsubseteq_R f) \rightarrow (\forall n \geq 1.\exists R.f^n \sqsubseteq_R f).$$

Compared to Yggdrasil [66], which assumes that failure happens only once when verifying the idempotence of sequential recovery procedures, Flux targets a different setting — concurrent vs. sequential. Yggdrasil proves that if the execution with one retry produces the same system state and return value as the retry-free execution, then the execution with arbitrary times of retries also produces the same system state and return value. This approach ignores intermediate system states. It only considers the system state and return value when the function finishes because intermediate system states for sequential functions are not observable to clients. However, under concurrency, we should consider intermediate system states. We need to define and prove failure reduction based on simulation relation $\sqsubseteq_R$.

5 Implementation

Flux builds a verifier to automatically verify idempotence simulation for each function based on failure reduction. It models the execution of a function with symbolic traces generated by symbolic execution. Each trace represents a feasible execution path and records the path condition, events, and database operations. The verifier can only handle Java applications because Flux builds the verifier by extending a symbolic execution engine for Java [61]. However, our definition and verification method for idempotence consistency is not specific to any particular programming language.

When functions invoke third-party services, Flux mandates that developers explicitly indicate whether these services have side effects. In particular, developers provide a vector of service names and a corresponding bit vector, where each bit indicates whether the corresponding service has side effects. For instance, developers should annotate the *random* function with a "0" since it has no side effects, whereas developers should annotate the *print* function with a "1".

Next, Flux handles unbounded loops by first converting functions into abstract syntax trees (ASTs) and identifying unbounded loops within them. Then, Flux replaces each unbounded loop that does not modify the shared state with an uninterpreted function. It further identifies all variables modified within the loop and assigns the return value of the uninterpreted function to these variables. For each unbounded loop that modifies the shared state, Flux partitions the code into three parts via ASTs and checks them (Section 4.3).

Algorithm 1: Workflow of the Verifier

```
Input: A function set F, a function f ∈ F, a string vector
  services of the names of services, and a bit vector bv
  indicating whether each service has side effects.
Output: The verification result of f.
Verify(F, f, services, bv):
  R := GenRelyCond(F)
  T := TracesNoRetry(f, R, services, bv)
  T_r := TracesWithRetry(f, R, services, bv)
  foreach ⟨t_r, pc_r⟩ ∈ T_r:
    if ¬HasSimulatedTrace(⟨t_r, pc_r⟩, T) then
      return false
  return true
HasSimulatedTrace(⟨t_r, pc_r⟩, Traces):
  foreach ⟨t, pc⟩ ∈ Traces:
    if CheckSimulation(t_r, pc_r, t, pc) then
      return true
  return false
```

Algorithm 1 shows the verification algorithm. The goal is to prove the simulation relation between f^1 defined in Section 4.4 and f. First, *GenRelyCond* generates the rely condition R by symbolically executing all functions in F. Based on R, *TracesNoRetry* generates all possible symbolic traces T for f via another symbolic execution. *TracesWithRetry* returns all possible symbolic traces T_r for f^1. Then, for every trace $t_r \in T_r$, *HasSimulatedTrace* checks whether there exists a trace in T that can simulate t_r. The initial path condition of symbolic execution is true, which does not contain any constraints on database states and function arguments. Therefore, if Flux can find a retry-free trace for each trace t_r, the retry-free trace is feasible, and the idempotence simulation holds for any possible database states and arguments.

5.1 Generating Symbolic Traces

Each trace is an ordered list with a path condition. Every list element includes the following fields: step id, event, database operation, and the post-condition after the step. The step id identifies the atomic step causing state transition. The event produced by the step has a type, some arguments, and a return value, which are symbolic expressions or constants. Flux considers three types of events for the execution of a serverless function: function invocation, function response, and third-party services with side effects. Developers specify whether a third-party service has side effects via the bit vector *bv* in Algorithm 1 (line 3). The fields of the operation include the type (*optype*), the argument (*oparg*), and the result of the operation (*opret*), where *oparg* and *opret* are symbolic expressions or constants. As described in Section 4.2, Flux models the semantics of each API as a sequence of read or write operations. Post-conditions can help model the return value of read operations. For example, D[k]=c implies that the results of the subsequent read operations on k must be c. Flux adds these propositions about return values of read operations to the path condition.

GenRelyCond(F) symbolically executes all functions in F and returns the rely condition R. As illustrated in Section 4.2, Flux identifies three kinds of data. If an operation modifies the data indexed by a symbolic key, Flux assumes that the operation can change arbitrary data in the database. If an operation writes a symbolic value into the data, Flux uses the path condition to infer whether it is a constant or an arbitrary value. Users can also annotate that some variables in a serverless function are unique. That means other concurrent instances cannot access the data indexed by these unique variables. For example, developers can annotate *receiptId* in Figure 1 to be unique to indicate that other concurrent instances cannot write the receipt created by the current instance.

5.2 Checking Idempotence Simulation

Algorithm 2: Checking Idempotence Simulation

```
1  CheckSimulation(t_r, pc_r, t, pc):
2    premise := pc_r ∧ pc
3    pass := CheckWithPremise(t_r, t, premise)
4    return pass
5
6  CheckWithPremise(t_r, t, premise):
7    if t_r.empty() then
8      return true
9    step := t_r.subtrace(0, 1)
10   foreach n from 0 to t.size() − 1:
11     nsteps := t.subtrace(0, n)
12     pass := CheckStep(step, nsteps, premise)
13     if !pass then
14       continue
15     next_premise := UpdatePremise(step, nsteps, premise)
16     next_t_r := t_r.subtrace(1, t_r.size())
17     next_t := t.subtrace(n, t.size())
18     pass := CheckWithPremise(next_t_r, next_t, next_premise)
19     if pass then
20       return true
21   return false
```

CheckSimulation in Algorithm 2 determines if two traces, t_r from f^1 and t from f, satisfy the idempotence simulation. Their associated path conditions are pc_r and pc. Specifically, *CheckSimulation* tries to construct a mapping from every step in t_r to n $(n \geq 0)$ steps in t such that the n steps can simulate the single step. The existence of such a step mapping can imply idempotence simulation.

CheckWithPremise recursively checks all possible step mappings. It first uses *CheckStep* (Line 12) to check if the n steps in t (*nsteps*) can simulate the first step in t_r (*step*). If the check fails, it increases n to enumerate other possible mappings. Otherwise, it continues to check the subsequent steps in t_r in a recursive way (Line 18). To reason the simulation between *step* and *nsteps*, *CheckStep* requires that the write operations in *step* and *nsteps* result in the same database state. It means that every write operation in *nsteps* should have the same parameters as the write operation in *step* under the proposition of *premise*, where *premise* is the conjunction of the path conditions associated with t and t_r. Specifically, two symbolic parameters, p_1 and p_2, are equivalent under *premise* if $premise \rightarrow (p_1 = p_2)$ is true. Flux leverages an SMT solver to check this first-order logic formula. If *step* does not record any write operations, *nsteps* should also contain no write operations. Besides, when *step* has an event, such as invoking a third-party service with side effects and function response, Flux requires that *nsteps* contains the same event. After the check succeeds, Flux updates *premise* with *UpdatePremise*, which maintains the relations among symbolic variables. *UpdatePremise* has three parameters, including *step*, *nsteps*, and *premise*. If operations in *step* and *nsteps* read the data indexed by the same key in databases, *UpdatePremise* adds a proposition to *premise* that these operations return the same value. Otherwise, it does nothing.

Algorithm 2 enumerates all possible mappings, which introduces heavy verification burdens. Flux proposes a heuristic algorithm based on the observation that f^1 and f are almost the same, except that the platform may re-execute f^1. Thus, for each step s_i^1 of f^1 before the retry, Flux tries to map it to the ith step s_i of f. For each step s_j^1 of f^1 after the retry, if f^1 has executed s_j^1 before the retry, then Flux maps it to a nop step, a step that does nothing. Otherwise, it maps s_j^1 to a step s_k in f such that s_k can simulate s_j^1. This only constructs and checks one mapping instead of enumerating all possible mappings, which may miss the correct mapping. If the constructed mapping does not work, Flux will randomly sample other mappings to reduce false positives. However, the method causes no false positives in the evaluation.

Example. After logging all operations except for the *put* operation, the *payment* function will have no idempotence issues. We also need to log *generateId*, which always returns the same value on retry. First, *GenRelyCond(F)* returns a rely condition that other concurrent function instances can arbitrarily change *balance* and *discount* in the database because

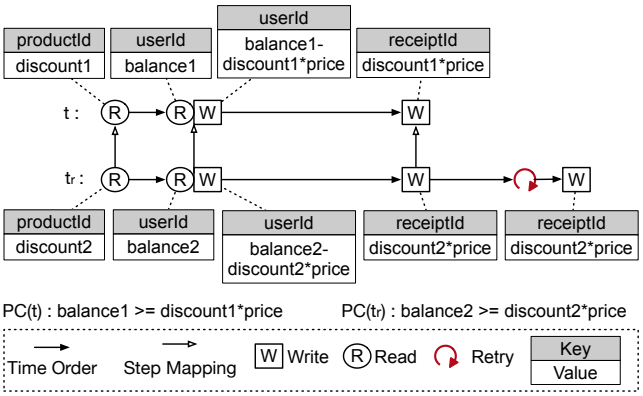


Figure 7: The example of verifying *payment* in Figure 1 when logging all database operations except for *put* (line 9). Variables, such as *discount1*, have symbolic values. PC(t) is the path condition of the trace t.

the function writes them by symbolic values. Second, Figure 7 shows that t_r is a symbolic trace produced by retrying *payment*[1] after it generates *receiptId*, while t is a symbolic trace produced without retries. Since Flux models *update* (line 4) as a read and a write operation executed atomically, there is no arrow between them. The post-conditions in traces are true. Third, Algorithm 2 finds a proper mapping from every step in t_r to n steps in t and returns true. According to the heuristic algorithm in the previous paragraph, there are three steps in t_r before the retry. Flux maps each of them to one step in t, accordingly. For instance, the first step in t_r corresponds to executing *get* (line 2). Flux maps it to the first step in t, corresponding to the execution of *get* (line 2). The last step in t_r corresponds to the *put* (line 9) executed in the first execution of *payment*[1]. Thus, Flux maps this step to a nop step. The first step in t can simulate the first step in t_r because they do not modify the database state and produce no events. Since the start database states of the two steps are the same, the return values of their read operations are the same, which means *discount1*=*discount2*. The second step in t can simulate the second step in t_r because they change *balance* to the same value with no events. Due to a similar reason, the third step in t can simulate the third step in t_r. Because of logging, *get* on retry in t_r still returns *discount2* in the first execution, while *update* on retry in t_r does nothing and returns true. The *generateId* function also returns the same identifier. Flux can use a nop step to simulate the last step in t_r because it is a useless write that does not change the data in the receipt. Since the function has no return value, we omit the function response event. Other traces of *payment* under retry can also pass the verification. Therefore, *payment* with these logs has no idempotence issues.

Soundness and Completeness. The verifier is sound and incomplete. Soundness means the verifier will not overlook any idempotence issues, which the theorems in Section 4 can imply. Incompleteness means some idempotence-consistent

```
1 void unsupportedLoop(key, n) {
2   value := get("Data", key);
3   while(value % n != 0) {
4     put("Data", key, value + 1);
5     value := get("Data", key);
6   }
7 }
```

Figure 8: An example of an unbounded loop unsupported by Flux.

applications cannot pass the verification. Note that although the verifier is incomplete, it can still ensure idempotence consistency and reduce the performance overhead of logging. The verifier is incomplete and will introduce false positives in the following cases. First, an application is idempotence-consistent, but individual functions do not satisfy idempotence simulation. For example, an application incorporates two functions, f_1 and f_2. They blindly write different values to the same record R_a, while no functions will read it. Consider the interleaving of f_1.write(R_a, 1) → f_2.write(R_a, 0) → f_1.retry() → f_1.write(R_a, 1). In this example, functions flip R_a's value twice due to the retry. Nevertheless, since no functions read the record R_a, clients will not observe that two flips occur. Idempotence consistency still holds. However, Flux will consider these two writes as non-idempotent because they fail to follow idempotence simulation and will log each of them; Second, an application contains unbounded loops with write operations such that the parameters of issued write operations can be different between normal execution and retry. For example, the unbounded loop in Figure 8 uses the data read from the database to be the parameters of write operations and the loop condition. As a result, the parameters of write operations and the number of loop iterations may change on retry, which does not satisfy the requirements of Theorem 2. Section 4.3 depicts two specific types of unbounded loops that can be handled by Flux. Third, when examining the idempotence simulation of a function f, Flux constructs a step mapping using a heuristic approach instead of enumerating all potential mappings. The heuristic runs under the assumption that if f fails before a specific operation op, f will execute op on retry. When the assumption does not hold, the heuristic may not work, and Flux may miss correct mappings. Last, Flux does not generate all possible rely, pre- and post-conditions, which may impede the verification capability of idempotence simulation.

6 Advisor

Advisor identifies idempotence-violating operations based on a brute-force algorithm and a heuristic algorithm. The brute-force algorithm first enumerates all possible operation sets. Second, for every set, it invokes the verifier to prove the function after ensuring exactly-once execution of each operation in the set via logs. Flux models the exactly-once execution of an operation by adjusting the generated traces rather than modifying functions. Specifically, it deletes each trace element that records a retried write operation in the set.

Table 1: An example of executing the heuristic algorithm of advisor.

`get`	0	1	1
`update`	1	0	1
`put`	1	1	0
`Verify`(F, f)	False	False	True

Furthermore, it appends a formula to the path condition for each retried read operation in the set, indicating that the retried operation returns the same result as the initial execution. Third, among all operation sets that enable the function to pass the verification, Flux selects the set that incurs the least performance cost. Flux estimates the performance cost via the evaluation results derived from the adopted logging mechanism. This algorithm enumerates $O(2^n)$ possible operation sets, where n is the number of operations.

To reduce the complexity, we propose a heuristic algorithm checking only $O(n)$ sets. Initially, Flux logs all operations in the function. Then Flux gradually removes the logs of operations one by one. If the function can pass the verification after removing a log, advisor will permanently remove the log. Otherwise, the operation is idempotence-violating, and advisor preserves the log. Table 1 shows how to apply the heuristic to *payment* in Figure 1. Assume that advisor has logged *generateId*. "0" denotes that advisor logs the operation. "1" denotes that advisor does not log the operation. Advisor first removes the log of *get* (get=0) but Verify(F, f) returns $false$. Therefore, advisor preserves the log for idempotence. Due to the same reason, advisor does not remove the log of *update*. Last, Verify(F, f) indicates removing the log of *put* does not break the idempotence. Therefore, advisor does not log it.

The incompleteness of the verifier may result in advisor being unable to detect certain redundant logs. Nonetheless, Flux can still diminish logging overhead for functions while ensuring idempotence consistency. Specifically, for an unbounded loop that the verifier cannot handle, advisor logs all operations before and within the loop except for read operations on read-only data. Then advisor directly addresses the code after the unbounded loop.

7 Evaluation

We aim to answer the following questions: 1) How effective is the verifier? 2) How long does it take for advisor to identify idempotence-violating operations? 3) How much performance benefit does Flux bring?

7.1 Experimental Setup

We evaluate the execution time of the verifier and advisor on a desktop running Ubuntu 18.04, which has an Intel Core i7-8700 processor and 15GB DRAM. Additionally, we evaluate the performance of serverless applications on multiple AWS servers.

We compare our system with Beldi [73] and Boki [45], which logs all operations. Although some other systems [44, 69] also guarantee idempotence for serverless applications, we focus on comparing with Beldi and Boki because they are state-of-the-art. It is worth noting that while Flux aims to provide idempotence consistency, Beldi and Boki go further by ensuring transactional properties as well. To ensure fairness in our comparison, we do not utilize the transaction mechanisms provided by Beldi and Boki, as our focus is on idempotence assurance alone. Therefore, the advantage that Flux may have over Beldi and Boki in terms of performance does not result from its lack of guarantee for transactional properties.

We store the data of applications in DynamoDB [15]. We run Beldi on AWS Lambda [16] with 1GB DRAM for each instance and collect performance results via AWS CloudWatch. Boki provides its own serverless platform. We deploy Boki according to the evaluation environment in its paper [45]. Boki's serverless platform can report the latencies of functions but cannot report their throughputs precisely.

We use wrk2 [1] as the load generator, which runs on an m5.2xlarge instance for Beldi and a c5d.xlarge instance for Boki. We adapt representative applications from the AWS serverless applications repository [3], a GitHub repository (10.9k stars) [9], popular benchmarks [8, 10, 29], and applications commonly used in papers about serverless computing [5, 6]. They have covered diverse real-world scenarios of serverless computing, such as image processing and web applications. We choose the applications with at least 1,000 deployments in AWS serverless application repository [3]. We skip some applications since they are micro-benchmarks or stateless (no database operations). Table 2 summarizes the characteristics of these applications. **Type-I** applications satisfy idempotence consistency, while **Type-II** applications do not. Although most applications in Table 2 have fewer than a thousand lines of code, they remain representative because serverless platforms typically impose restrictions on code size and running time [2, 4, 7]. For instance, existing work [75] that adapts web applications to serverless platforms needs to largely reduce the application's code size.

Flux's approach is orthogonal to specific programming languages. However, its implementation depends on a Java symbolic execution engine [61] for program analysis. We manually port applications to Go with the same semantics for a fair performance comparison with Beldi and Boki, which target Go applications. We choose to implement a verifier for applications in Java rather than Go because Java is a more frequently utilized language in developing serverless applications than Go [11].

7.2 Verification Efficacy

Table 2 shows that the verifier identifies all 12 applications with idempotence issues. All issues are previously unknown. Developers have confirmed 8 issues among them. The verifier works for all applications except SPECjbb, which has

Table 2: The characteristics of 27 serverless applications. The applications with † have unbounded loops. **LoC** indicates lines of Java code. **C/S** indicates whether functions run sequentially (S) or concurrently (C). **#F**, **#I**, and **#N** indicate the number of functions, functions without idempotence issues, and functions with issues. **#R/#W** and **#S** indicate the number of read/write operations and idempotence-violating operations. **VTime**, **ATime(H)**, and **ATime(B)** indicate the execution time (in seconds) of the verifier, advisor using the heuristic algorithm, and advisor using the brute-force algorithm.

Type	Application	LoC	C/S	Selected	#F	#I	#N	#R	#W	#S	VTime	ATime(H)	ATime(B)
I (15)	Data Analysis [10] †	356	S	✓	7	7	0	4	3	0	5.45		
	Image-Processing [10]	435	S		5	5	0	0	4	0	3.24		
	Mapreduce [8] †	250	S		3	3	0	3	1	0	2.06		
	FaaSImage [8]	193	S		1	1	0	1	9	0	104.31		
	Video [8] †	40	S		1	1	0	1	1	0	2.19		
	Image-Resizer [9]	92	S		1	1	0	1	1	0	2.40		
	Replicator [9]	59	S	✓	1	1	0	1	1	0	2.40		
	Receive-Email-Body [9]	58	S		1	1	0	1	0	0	0.74		
	Fetch-And-Store [9]	66	S		1	1	0	0	1	0	1.49		
	FFmpeg [9] †	49	S		1	1	0	1	-	-	2.23		
	DynamoDB-backup [9] †	26	S		1	1	0	1	-	-	2.44		
	Lambda-Image-Resizer [3]	123	S		1	1	0	1	0	0	1.12		
	Uploader [3]	101	S	✓	1	1	0	2	1	0	1.95		
	FFmpeg-Lambda-Layer [3]	86	S		1	1	0	1	1	0	2.30		
	Image-magick [3]	86	S		1	1	0	1	1	0	2.49		
II (12)	SPECjbb2015 [29] †	1,861	C	✓	9	5	4	-	-	-	Timeout	6.23	11.87
	Alexa [10] †	89	C	✓	10	9	1	1	1	1	2.13	2.60	4.25
	Hotel [5] †	714	C	✓	10	8	2	7	5	3	2.26	17.70	68.94
	Media [5]	486	C		7	6	1	1	7	7	14.79	86.04	344.77
	Pynamodb-S3-URL [9]	224	C		6	2	4	6	9	9	9.91	22.85	69.71
	Rest-API [9]	135	C		4	1	3	2	2	3	4.90	5.98	23.72
	GraphQL [9]	80	C	✓	1	0	1	1	1	1	2.07	4.34	12.17
	Mongodb-Atlas [9]	83	S		1	0	1	0	1	1	2.28	14.25	72.67
	Express [9]	76	C		1	0	1	1	1	1	1.90	1.92	4.28
	Flask [9]	34	C		1	0	1	1	1	1	2.31	2.80	3.64
	Save [3]	62	S		1	0	1	0	1	1	2.14	5.61	12.57
	HttpEP [3]	105	C	✓	1	0	1	1	3	3	2.42	11.73	44.5

unbounded loops unsupported by the verifier. However, all functions in SPECjbb that cannot be verified have idempotence issues. Thus, the verifier does not introduce false positives for SPESjbb. Nonetheless, it is worth noting that false positives are still possible for other applications.

The 21 non-idempotent functions detected by Flux in 12 applications result in various bug patterns and outcomes due to incorrectly repeated write operations on retry. First, the database state is inconsistent with user expectations. For instance, the idempotence violation in SPECjbb causes duplicate balance deductions. Second, the value responded to clients is inconsistent with the database state. An example is an IoT application called Alexa, where a function successfully modifies device configuration but returns "failed" instead of "success" to clients. Third, a single write operation may update multiple records on retry, resulting in duplicated records with identical content. For example, the *PlaceOrder* function in the Hotel benchmark always places a new order with a random identifier on each retry, resulting in duplicate records. Last, concurrent functions may observe the inconsistent shared state. An example is the Media benchmark, which relies on a counter in the database to perform synchronization among concurrent functions. However, the *ComposeReview* function will falsely increase the counter due to retry, leading to false synchronization among concurrent instances.

To test the scalability of the verifier, we run micro-benchmarks with increasing verification complexity. When the number of branches in a single function increases, the verification time increases exponentially, as the number of traces also increases exponentially. Verifying a function with 16 branches takes 1441.48 seconds. When we increase the number of database operations, functions, or LoC, the verification time increases linearly because the number of generated traces increases linearly. Figure 15 in Appendix D presents more details. Note that the verification time does not affect the execution time of applications.

7.3 Performance of Advisor

For the second question, Table 2 shows the execution time of advisor with two different algorithms. Compared to the brute-force algorithm, the heuristic algorithm achieves up to 4× smaller search space, which cuts down 80.39% of the execution time of advisor. Although the heuristic algorithm does not guarantee finding the minimum operation set, the evaluation shows that it finds the same set as the brute-force algorithm in practice. Although SPECjbb2015 has unbounded loops unsupported by the verifier, advisor can still handle it with the method described in Section 6.

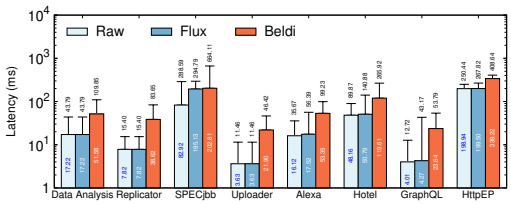
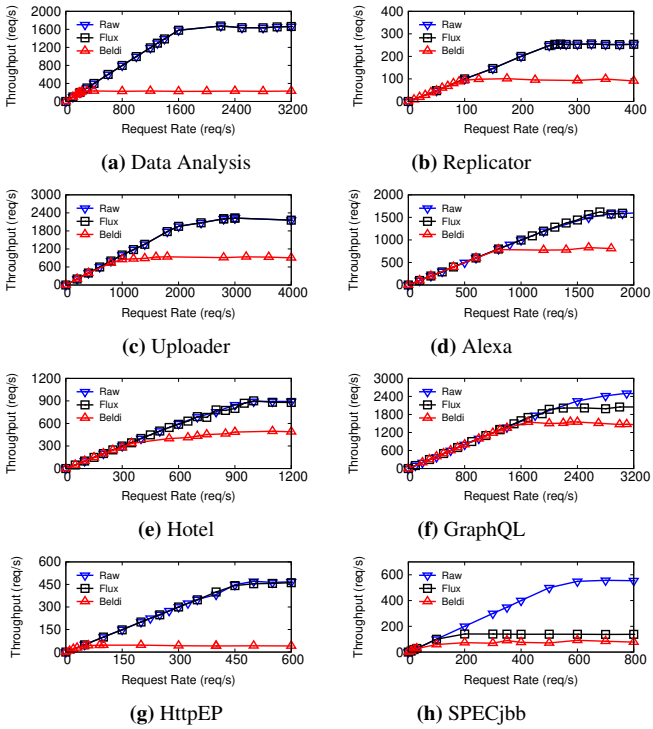


Figure 9: The median (box) and 99% tail latency (whisker) of Raw, Flux, and Beldi for eight applications. The y-axis is in the log scale.



(a) Data Analysis **(b)** Replicator

(c) Uploader **(d)** Alexa

(e) Hotel **(f)** GraphQL

(g) HttpEP **(h)** SPECjbb

Figure 10: The throughput of each serverless application under different configurations with increasing request rates.

7.4 Performance of the Applications

To answer the third question, we run applications under four configurations: Raw, Flux, Boki, and Beldi. Raw means running applications without any logs, which may be non-idempotent. Flux uses existing mechanisms to log operations identified by advisor. Wrk2 runs for 7 minutes to generate random requests. We will show the results of 8 representative applications marked by ✓in Table 2. Since Type-I applications are idempotence-consistent, Flux removes all logging operations. For Type-II applications, Flux reduces up to 99.47% of logging operations during execution compared to Beldi and Boki.

7.4.1 Flux vs. Beldi

Latency. Figure 9 shows the results on AWS Lambda. Flux poses no logging overhead over Raw for Type-I applications. Compared with Beldi, which logs all operations, Flux brings 2.5× ~ 6× performance improvement. For Type-II applications, Flux can avoid logging some operations. As a result, it achieves up to 5.5× performance improvement over Beldi.

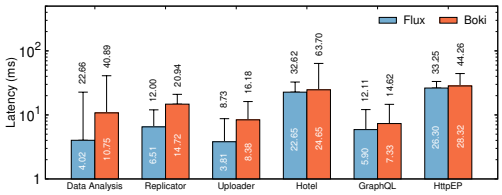


Figure 11: The median (box) and 99% tail latency (whisker) of Flux and Boki for six applications at 1 RPS.

Throughput. Figure 10 shows that Flux achieves the same peak throughput on Type-I applications as Raw and up to 7.36× higher peak throughput than Beldi. For Type-II applications, Flux can avoid part of logging operations. Therefore, it has up to 80% higher peak throughput than Beldi, except for HttpEP. For HttpEP, Flux has 10× higher peak throughput than Beldi because Flux avoids logging a scan operation that can return massive data, which reduces much overhead under concurrency.

7.4.2 Flux vs. Boki

Boki [45] is another system to provide idempotence for serverless applications. The comparison can also demonstrate that Flux is general enough to be independent on a specific serverless platform. Because of Boki's limitations, it does not guarantee idempotence for read-modify-write database operations [13]. Thus, we do not evaluate Alexa and SPECjbb2015 on Boki.

Figure 11 shows that at 1 RPS, Flux reduces the median latency by up to 62.6% compared to Boki. The performance of Boki is better than Beldi because Boki designs a more efficient logging mechanism. Under high concurrency, the logging mechanism of Boki introduces more overhead. We further evaluate the latency at 800 RPS. Flux reduces the median latency by up to 82.5% and the p99 latency by up to 69.0% for Type-I applications compared to Boki because Flux introduces no logs. For Type-II applications, Flux reduces the median latency by up to 88.4% and the p99 latency by up to 72.4%. Flux achieves 8.7× lower median latency than Boki on HttpEP because Flux avoids logging an expensive "scan".

7.4.3 Performance of the Java Applications

Since Beldi and Boki only support Go applications, we compare the performance of Java applications on AWS Lambda under the configurations of Flux and Raw. We implement the logging mechanism via transactions [73]. At 1 RPS, Flux achieves a tail latency up to 22.6% higher than Raw due to the additional logging overhead. However, despite better performance, Raw cannot guarantee idempotence consistency and may cause incorrect execution results.

8 Related Work

Verification of Idempotence. Table 3 summarizes the major differences between Flux and prior works that can verify

Table 3: Main differences between Flux and prior works. FSCQ-based works use a method akin to FSCQ [28] to prove the idempotence. Partial protection means ensuring idempotence consistency while reducing unnecessary logs.

	Definition of Idempotence		Verification Method of Idempotence				Protection
	Support Concurrency	Target Serverless Applications	Verification Targets	Support Concurrency	Automated Verification	Unbounded Loop	Partial Protection
Flux	✓	✓	Implementation	✓	✓	✓(Partially)	✓
Ramalingam et al. [62]	✓	✗	Protocol	✓	✗	–	✗
Jangda et al. [44]	✓	✓	Protocol	✓	✗	–	✗
Yggdrasil [66]	✗	✗	Implementation	✗	✓	✗	–
FSCQ-Based Works [23–25, 27, 28]	✗	✗	Implementation	✗	✗	✓	–

idempotence. Jangda et al. [44] formalizes the semantics of serverless computing and ensures idempotence with transactions. There are two main differences between Jangda et al. and Flux. For the idempotence definition, idempotence consistency is more relaxed than the idempotence provided by Jangda et al., as it allows more concurrent schedulings. Jangda et al. tries to model serverless computing with naive semantics to conceal the low-level details of serverless function execution, such as concurrency and warm-start. It requires the platform to process concurrent requests in the same way as processing a single request at a time without concurrency or retries. Consequently, it necessitates atomicity, serializability, and exactly-once execution. Besides, Jangda et al. focuses on verifying protocols instead of source code. To ensure idempotence, it protects the entire function with serializable transactions and uses logs to ensure that the transaction only commits once. However, using transactions is not optimal since some applications may not require transaction semantics, resulting in redundant protection and performance cost.

FSCQ [28] and DFSCQ [27] verify the crash safety of file systems via Crash Hoare Logic (CHL). CHL requires developers to manually write pre-conditions, post-conditions, and crash conditions to specify crash safety. Under crash, CHL proves that the program state always satisfies crash conditions after a crash happens at any time. It defines the idempotence of a recovery program to be that crash conditions imply pre-conditions. Perennial [23], GoJournal [24], and DaisyNFS [25] achieve significant progress in the verification of concurrent crash-safe systems. They adopt a similar approach as FSCQ to verify the idempotence of a recovery program. Thus, they cannot realize automated verification. Different from them, Flux focuses on automated verification without human effort.

Recent SMT-based verification approaches [26,39,66] have solved many issues of the automated verification of storage systems. Yggdrasil proposes a new correctness definition for sequential file systems — *crash refinement*. It means that for any disk state produced by the implementation with crash recovery, the specification can also produce the same disk state. The definition is amenable to automated verification. Yggdrasil also verifies the idempotence of recovery functions. Unfortunately, it verifies sequential functions rather than concurrent functions. Others [26, 39] also verify sequential functions.

Ramalingam et al. [62] formally defines idempotence in the general distributed setting, proving that logging each operation can guarantee this property. It has two main differences from Flux. First, for the definition of idempotence, both Ramalingam et al. and Flux require that any execution with retries has the same observable behavior as another normal execution. However, Ramalingam et al. targets a general distributed setting. The observable behavior only includes function invocation and response. In contrast, Flux also considers database states. As a result, for stateful serverless applications, Ramalingam et al.'s definition may allow inconsistent database states on retry as long as functions do not return database states to clients. This anomaly is critical if clients directly access database states for analysis or other purposes. In contrast, Flux's definition prohibits inconsistent database states and can prevent such anomalies. Second, for verification and protection, Ramalingam et al. manually proves the protocol of logging each operation and presents a compiler that automatically logs each operation. Flux focuses on verifying the implementation of applications and its advisor only logs necessary operations.

Crash-Only Software. Previous work [20] comprehensively analyzes the requirements for crash-only software, a program that can crash safely and recover quickly via retries. Our work can assist in revising these requirements. Specifically, serverless applications fulfill most of these requirements, such as explicit boundaries around what is retried and dedicated storage for non-volatile data. However, our work only mandates that all functions satisfy idempotence consistency instead of necessitating that every request is idempotent, as stated in previous work [20]. This distinction results in fundamental differences in several aspects. First, idempotence pertains to a single component, while idempotence consistency pertains to the entire application comprised of multiple components. Second, while idempotence requires a function's execution result to remain the same despite retries, idempotence consistency only requires the existence of an execution without retries for each execution with retries such that they produce the same result. For example, a read-only function

may not satisfy idempotence because concurrent modifications to the database state can result in different return values with and without retries. However, read-only functions do not violate idempotence consistency. In addition to serverless functions, other applications with "imperfectly crash-only" code can also refer to the requirements revised based on Flux.

Research on Serverless Computing. Some systems [44, 45, 62, 69, 73] focus on ensuring idempotence via runtime mechanisms. Different from them, Flux focuses on verifying idempotence consistency. Furthermore, combining them with Flux can ensure idempotence efficiently. Some other systems [21, 30–32, 46, 49, 51, 56, 63, 65, 71, 72, 74] target other things, such as startup time.

Consistency Model. There are many consistency models [14], which define the permitted execution order under concurrency. Idempotence consistency specifies the expected behavior of concurrent systems when failure happens, which is the main difference from existing consistency models.

9 Discussion

Serverless Applications vs. Other Applications. *Idempotence* [43] is a property that is important not only for serverless applications but also for programs that use retry-based fault tolerance approaches, such as RPC-based distributed systems [41, 50], AI systems [38], and even some intermittent systems [70]. Thus, it is worth considering *whether we can apply Flux to programs beyond serverless applications.* The answer is *yes* and *no*.

One of the key contributions of Flux is formally defining idempotence consistency, which is general enough for various scenarios. For instance, different RPC handlers in distributed systems may concurrently manipulate shared states. Repeated state updates due to failures and retries could impair data consistency under concurrency. Specifically, we successfully detect an issue in HDFS according to the definition of idempotence consistency, which the community has confirmed [41]. When the system retries NameNode RPC of *ClientProtocol.truncate*, it may truncate a file multiple times, which will potentially cause data loss if another RPC simultaneously updates the same file. However, this issue will not happen under sequential execution.

However, using Flux's method to verify other systems poses many challenges. First, the automated verification algorithm mainly focuses on serverless applications. We need to redesign it for other scenarios. For example, Flux presumes that shared states are solely key-value pairs stored in NoSQL databases. However, shared states could be file descriptors, shared variables, or global configurations in distributed systems. Programmers must reinterpret them across various scenarios. Additionally, Flux automatically constructs pre-, post-, and rely conditions based on the NoSQL interface assumption, necessitating a re-examination of the construction algorithm based on other modeling methods of states. Second, Flux's implementation targets serverless applications. For example, Flux currently only supports Java programs and DynamoDB, which are commonly used by serverless applications. Engineering effort is necessary to support other languages and storage services. Furthermore, advisor in Flux relies on existing logging mechanisms for serverless applications to fix idempotence issues. Therefore, the new scenario should also provide mechanisms to ensure exactly-once execution of operations on shared states. Failure to do so limits Flux's potential to identify and fix idempotence issues via advisor.

Idempotence Consistency vs. Atomicity. Although atomicity is vital for fault tolerance, Flux does not guarantee it. Atomicity is not mandatory for some applications to sustain fault tolerance, as they may permit other functions to see partial updates. Furthermore, it is essential to emphasize that Flux is orthogonal to approaches that ensure atomicity. An application that necessitates atomicity can still utilize Flux to guarantee idempotence consistency and minimize logging overhead.

Although Flux does not verify atomicity and some other transactional properties, extending our approach to verify these properties is an intriguing research direction. Some verifiers [22–25, 37, 53, 54, 76] target transactional properties but may not consider retries. After ensuring idempotence consistency based on Flux, we can prove the transactional properties of functions without considering failures and retries. Therefore, we can combine Flux and these verifiers by ensuring idempotence consistency via Flux and then proving other properties with these verifiers.

Automated Verification vs. Interactive Verification. We adopt symbolic execution engines to develop Flux, which requires less manual effort than using interactive theorem provers. [37]. Additionally, although interactive theorem provers can handle unbounded loops by crafting loop invariants, finding proper loop invariants is notoriously difficult.

10 Conclusion

This paper presents Flux, the first toolkit that can automatically verify and help ensure the idempotence consistency of serverless applications. It guarantees idempotence consistency via logs while reducing unnecessary logging overhead.

Acknowledgment

We sincerely thank the anonymous reviewers for their valuable comments. We are especially grateful to our shepherd, George Candea, whose reviews and suggestions largely improved our work. This work is supported by the National Natural Science Foundation of China (No. 62132014 and 62272304), the Fundamental Research Funds for the Central Universities, and the HighTech Support Program from Shanghai Committee of Science and Technology (No. 20ZR1428100). Zhaoguo Wang (zhaoguowang@sjtu.edu.cn) is the corresponding author.

References

[1] A Constant Throughput, Correct Latency Recording Variant of wrk. https://github.com/giltene/wrk2.

[2] AWS Lambda Enables Functions That Can Run up to 15 minutes. https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/?nc1=h_ls,.

[3] AWS Serverless Application Repository. https://serverlessrepo.aws.amazon.com/applications.

[4] Azure Functions Hosting Options. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale,.

[5] Beldi. https://github.com/eniac/Beldi.

[6] Benchmark Workloads of Boki. https://github.com/ut-osa/boki-benchmarks.

[7] Cloud Functions Execution Environment. https://cloud.google.com/functions/docs/concepts/exec,.

[8] Functionbench. https://github.com/kmu-bigdata/serverless-faas-workbench.

[9] Serverless Examples. https://github.com/serverless/examples.

[10] Serverlessbench. https://serverlessbench.systems/en-us/.

[11] State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[12] Uninterpreted Functions and Constants. https://microsoft.github.io/z3guide/docs/logic/Uninterpreted-functions-and-constants.

[13] Working with Items and Attributes - Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.AtomicCounters.

[14] Marcos K. Aguilera and Douglas B. Terry. The Many Faces of Consistency. *IEEE Data Eng. Bull.*, 39:3–13, 2016.

[15] Amazon. AWS Dynamodb. https://aws.amazon.com/dynamodb/.

[16] Amazon. AWS Lambda. https://aws.amazon.com/lambda/.

[17] Amazon. Build a CRUD API with Lambda and DynamoDB. https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-dynamo-db.html.

[18] Amazon. Make a Lambda Function Idempotent. https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/,.

[19] Andreas Blass and Yuri Gurevich. Inadequacy of Computable Loop Invariants. *ACM Trans. Comput. Logic*, 2(1):1–11, jan 2001.

[20] George Candea and Armando Fox. Crash-Only Software. In *9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003. USENIX Association.

[21] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, page 58–64, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying Concurrent Software Using Movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 306–322, Carlsbad, CA, October 2018. USENIX Association.

[23] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Huntsville, ON, Canada, October 2019.

[24] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: A Verified, Concurrent, Crash-safe Journaling System. In *15th USENIX Symposium on Operating Systems Design and Implementation*, pages 423–439. USENIX Association, July 2021.

[25] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS Concurrent and Crash-safe File System With Sequential Reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association.

[26] Yun-Sheng Chang, Yao Hsiao, Tzu-Chi Lin, Che-Wei Tsao, Chun-Feng Wu, Yuan-Hao Chang, Hsiang-Shang Ko, and Yu-Fang Chen. Determinizing Crash Behavior with a Verified Snapshot-Consistent Flash Translation Layer. In *14th USENIX Symposium on Operat-*

ing Systems Design and Implementation, pages 81–97. USENIX Association, November 2020.

[27] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Standard Performance Evaluation Corporation. Specjbb 2015. https://www.spec.org/jbb2015/.

[30] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 797–813, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Alexander Fuerst and Prateek Sharma. *FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching*, pages 386–400. Association for Computing Machinery, New York, NY, USA, 2021.

[33] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop Invariants: Analysis, Classification, and Examples. *ACM Comput. Surv.*, 46(3), jan 2014.

[34] Google. Google Cloud Functions. https://cloud.google.com/functions/.

[35] Google. Retrying Event-Driven Functions. https://cloud.google.com/functions/docs/bestpractices/retries,.

[36] Google. Stateful Serverless on Google Cloud with Cloudstate and Akka Serverless. https://cloud.google.com/blog/topics/developers-practitioners/stateful-serverless-on-google-cloud-with-cloudstate-and-akka-serverless.

[37] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 653–669, Savannah, GA, November 2016. USENIX Association.

[38] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[39] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 99–115. USENIX Association, November 2020.

[40] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[41] Apache HDFS. HDFS-16322. https://issues.apache.org/jira/browse/HDFS-16322.

[42] Hadoop HDFS. HDFS-7926. https://issues.apache.org/jira/browse/HDFS-7926.

[43] Pat Helland. Idempotence Is Not a Medical Condition: An Essential Property for Reliable Systems. *Queue*, 10(4):30–46, apr 2012.

[44] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.*, 3, October 2019.

[45] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Zhipeng Jia and Emmett Witchel. *Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices*, pages 152–166. Association for Computing Machinery, New York, NY, USA, 2021.

[47] C. B. Jones. Tentative Steps toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, oct 1983.

[48] Kafka. Kafka-5169. https://issues.apache.org/jira/browse/KAFKA-5169.

[49] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference*, pages 805–820. USENIX Association, July 2021.

[50] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. *Implementing Linearizability at Large Scale and Low Latency*, pages 71–86. Association for Computing Machinery, New York, NY, USA, 2015.

[51] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. FaasFlow: Enable Efficient Workflow Execution for Function-as-a-Service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782–796, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Hongjin Liang and Xinyu Feng. Modular Verification of Linearizability with Non-Fixed Linearization Points. *SIGPLAN Not.*, 48(6):459–470, jun 2013.

[53] Hongjin Liang, Xinyu Feng, and Ming Fu. A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 455–468, New York, NY, USA, 2012. Association for Computing Machinery.

[54] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–210, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Nancy Lynch and Frits Vaandrager. Forward and Backward Simulations. *Inf. Comput.*, 128(1):1–25, jul 1996.

[56] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. Orion and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[57] Microsoft. Designing Azure Functions for Identical Input. https://docs.microsoft.com/en-us/azure/azure-functions/functions-idempotent,.

[58] Microsoft. Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[59] Microsoft. What are Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[60] Peter W. OHearn. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, apr 2007.

[61] Java Pathfinder. Java Pathfinder. https://github.com/javapathfinder/.

[62] Ganesan Ramalingam and Kapil Vaswani. Fault Tolerance via Idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 249–262, New York, NY, USA, 2013. Association for Computing Machinery.

[63] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, New York, NY, USA, 2022. Association for Computing Machinery.

[64] Serverlessbench. Issue with Exactly-once Execution Semantic of Alexa. https://github.com/SJTU-IPADS/ServerlessBench/issues/6.

[65] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A Fast, Efficient, and Safe Serverless Framework Using VM-level post-JIT Snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[66] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Savannah, GA, November 2016. USENIX Association.

[67] Spark. Spark-6133. https://issues.apache.org/jira/browse/SPARK-6133.

[68] Spree. Spree. https://spreecommerce.org/.

[69] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, New York, NY, USA, 2020. Association for Computing Machinery.

[70] Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/o Dependent Idempotence Bugs in Intermittent Systems. *Proc. ACM Program. Lang.*, 3, oct 2019.

[71] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[72] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. InFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-Tolerant and Transactional Stateful Serverless Workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 1187–1204. USENIX Association, November 2020.

[74] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

[75] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. BeeHive: Sub-Second Elasticity for Web Services with Semi-FaaS Execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 74–87, New York, NY, USA, 2023. Association for Computing Machinery.

[76] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 259–274, New York, NY, USA, 2019. Association for Computing Machinery.

Appendix A Proof of Theorem 1

In this section, we prove that it is sufficient to verify the idempotence consistency of a function set F by proving that each function $f \in F$ satisfies idempotence simulation, which is Theorem 1 in the paper. Since our verification approach adopts existing compositional proof techniques in RGSim [53], our formal proof is similar to the proof in that paper.

Let's start by introducing some important concepts. When we have two functions running concurrently, we can consider them as one unit and denote their local state using the symbol $\sigma_1 \parallel \sigma_2$, where σ_1 represents the local state of the first function and σ_2 represents the local state of the second function. Using this notation, we can represent the overall system state as $\langle \sigma_1 \parallel \sigma_2, D\rangle$ when two functions run concurrently. We use $\langle \sigma_1 \parallel \sigma_2, D\rangle \xrightarrow{\alpha} \langle \sigma_1' \parallel \sigma_2, D'\rangle$ to denote that the system can take one step and change the system state from $\langle \sigma_1 \parallel \sigma_2, D\rangle$ to $\langle \sigma_1' \parallel \sigma_2, D'\rangle$, producing an event α (if any). During each execution cycle, the system can either advance the first function by one step or advance the second function by one step. It means that if there exists σ_1' and D' such that $\langle \sigma_1, D\rangle \xrightarrow{\alpha} \langle \sigma_1', D'\rangle$, then we can imply that $\langle \sigma_1 \parallel \sigma_2, D\rangle \xrightarrow{\alpha} \langle \sigma_1' \parallel \sigma_2, D'\rangle$. Similarly, if there exists σ_2' and D' such that $\langle \sigma_2, D\rangle \xrightarrow{\alpha} \langle \sigma_2', D'\rangle$, then we can imply that $\langle \sigma_1 \parallel \sigma_2, D\rangle \xrightarrow{\alpha} \langle \sigma_1 \parallel \sigma_2', D'\rangle$.

$\langle \sigma_1 \parallel \sigma_2, D\rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D\rangle$ denotes that the concurrent execution of two functions with the local state Σ_1 and Σ_2 can simulate the concurrent execution of another two functions with the local state σ_1 and σ_2 under the rely condition R. Here is the detailed definition of the simulation relation. We use $\langle \Sigma_1, D\rangle \overset{\alpha}{\Longrightarrow} \langle \Sigma_1', D'\rangle$ to denote that the system can take n ($0 \le n$) steps to change the system state from $\langle \Sigma_1, D\rangle$ to $\langle \Sigma_1', D'\rangle$, producing an event α (if any).

- For any local state σ_1' and database state D', if $\langle \sigma_1, D\rangle \xrightarrow{\alpha} \langle \sigma_1', D'\rangle$, then there exists a local state Σ_1' such that $\langle \Sigma_1, D\rangle \overset{\alpha}{\Longrightarrow} \langle \Sigma_1', D'\rangle$. If α is a *response* event, then $\langle \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_2, D'\rangle$. Otherwise, $\langle \sigma_1' \parallel \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_1' \parallel \Sigma_2, D'\rangle$. This condition represents the requirement for the simulation relation when the system advances the first function by one step.

- For any local state σ_2' and database state D', if $\langle \sigma_2, D\rangle \xrightarrow{\alpha} \langle \sigma_2', D'\rangle$, then there exists a local state Σ_2' such that $\langle \Sigma_2, D\rangle \overset{\alpha}{\Longrightarrow} \langle \Sigma_2', D'\rangle$. If α is a *response* event, then $\langle \sigma_1, D'\rangle \sqsubseteq_R \langle \Sigma_1, D'\rangle$. Otherwise, $\langle \sigma_1 \parallel \sigma_2', D'\rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2', D'\rangle$. This condition represents the requirement for the simulation relation when the system advances the first function by one step, which is similar to the first requirement.

- If $(D, D') \in R$, then $\langle \sigma_1 \parallel \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D'\rangle$.

We can extend the above definition to more than two functions.

To prove Theorem 1, we first prove the following two lemmas based on the above definitions.

Lemma 1 Assume there are four functions f_1, f_2, g_1, and g_2. Suppose that the execution of g_1 can simulate the execution of f_1 under the rely condition R, while the execution of g_2 can simulate the execution of f_2 under the rely condition R. Then we can imply that the concurrent execution of g_1 and g_2 can simulate the concurrent execution of f_1 and f_2 under the rely condition R. We use σ_1, σ_2, Σ_1, and Σ_2 to represent the local states of f_1, f_2, g_1, and g_2, respectively.

$$\begin{aligned}&\forall \sigma_1, \Sigma_1, \sigma_2, \Sigma_2, D.\\ &\quad (\langle \sigma_1, D\rangle \sqsubseteq_R \langle \Sigma_1, D\rangle) \wedge (\langle \sigma_2, D\rangle \sqsubseteq_R \langle \Sigma_2, D\rangle) \rightarrow\\ &\quad \langle \sigma_1 \parallel \sigma_2, D\rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D\rangle\end{aligned}$$

Proof

The premises include

$$\langle \sigma_1, D\rangle \sqsubseteq_R \langle \Sigma_1, D\rangle, \tag{1}$$

and

$$\langle \sigma_2, D\rangle \sqsubseteq_R \langle \Sigma_2, D\rangle. \tag{2}$$

The conclusion is

$$\langle \sigma_1 \parallel \sigma_2, D\rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D\rangle. \tag{3}$$

Below we prove the conclusion by *co-induction* on $\sqsubseteq_R$. According to the definition of $\sqsubseteq_R$ described above, the execution of two functions belongs to one of the following five cases.

- $\langle \sigma_1, D\rangle \xrightarrow{\alpha} \langle \sigma_1', D'\rangle$, where α is not a *response* event.

  According to the definition of $\sqsubseteq_R$, we need to prove that there exists Σ_1' such that $\langle \Sigma_1, D\rangle \overset{\alpha}{\Longrightarrow} \langle \Sigma_1', D'\rangle$ and $\langle \sigma_1' \parallel \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_1' \parallel \Sigma_2, D'\rangle$.

  From Equation (1), there exists Σ_1' such that

  $$\langle \Sigma_1, D\rangle \overset{\alpha}{\Longrightarrow} \langle \Sigma_1', D'\rangle, \tag{4}$$

  and

  $$\langle \sigma_1', D'\rangle \sqsubseteq_R \langle \Sigma_1', D'\rangle. \tag{5}$$

  Since $(D, D') \in R$, from Equation (2), we know

  $$\langle \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_2, D'\rangle. \tag{6}$$

  From both Equation (5) and Equation (6) we know

  $$\langle \sigma_1' \parallel \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_1' \parallel \Sigma_2, D'\rangle. \tag{7}$$

- $\langle \sigma_2, D\rangle \xrightarrow{\alpha} \langle \sigma_2', D'\rangle$, where α is not a *response* event. The proof is similar to the first case.

- $\langle \sigma_1, D\rangle \xrightarrow{\alpha} \langle \sigma_1', D'\rangle$, where α is a *response* event.

  According to the definition of $\sqsubseteq_R$, we need to prove that there exists Σ_1' such that $\langle \Sigma_1, D\rangle \xRightarrow{\alpha} \langle \Sigma_1', D'\rangle$, and $\langle \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_2, D'\rangle$.

  From Equation (1), there exists Σ_1' such that

$$\langle \Sigma_1, D\rangle \xRightarrow{\alpha} \langle \Sigma_1', D'\rangle. \tag{8}$$

  Since $(D, D') \in R$, we know

$$\langle \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_2, D'\rangle. \tag{9}$$

- $\langle \sigma_2, D\rangle \xrightarrow{\alpha} \langle \sigma_2', D'\rangle$, where α is a *response* event. The proof is similar to the third case.

- $(D, D') \in R$, which means that other concurrent functions change the database state from D to D'.

  According to the definition of $\sqsubseteq_R$, we need to prove that $\langle \sigma_1 \parallel \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D'\rangle$.

  From $(D, D') \in R$ and Equation (1), we know

$$\langle \sigma_1, D'\rangle \sqsubseteq_R \langle \Sigma_1, D'\rangle. \tag{10}$$

  From $(D, D') \in R$ and Equation (2), we know

$$\langle \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_2, D'\rangle. \tag{11}$$

  From both Equation (10) and Equation (11), we know

$$\langle \sigma_1 \parallel \sigma_2, D'\rangle \sqsubseteq_R \langle \Sigma_1 \parallel \Sigma_2, D'\rangle. \tag{12}$$

  Therefore, the conclusion Equation (3) is true. □

  Then we prove the following lemma.

Lemma 2 We use A_F to denote an automaton running functions in a function set F. For any function set F and automaton A_F, if every function in F satisfies idempotence simulation, then the following fact holds: from the same initial database state, every time the automaton A_{F^*} takes one step, A_F can take n steps ($n \geq 0$) such that they reach the same database state and produce the same event (if any).

Proof

The premise is that every function $f \in F$ satisfies idempotence simulation:

$$\begin{aligned} &\forall f \in F. \\ &\quad (\forall D, arg. \langle init(f^*, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle), \end{aligned} \tag{13}$$

where $init(f, arg)$ denotes the initial local state of running the function f with the argument arg, and we omit the existential quantifier on R. The conclusion is that the execution of F can simulate the execution of F^*.

We prove the conclusion by classifying every step taken by A_{F^*} into three cases.

- A_{F^*} creates a function instance to run a function f^*. Then, A_F can create an instance with the same identifier to run f with the same invocation arguments. Both of them do not change the database state and produce the same invocation event. Note that we treat f and f^* as the same in invocation events since their only difference is whether to be retried.

- A function instance takes one step. From Equation (13) and Lemma 1, we know that for any functions $f_1, f_2, \ldots \in F$, function arguments $arg_1, \ldots$, and shared state D,

$$\langle init(f_1^*, arg_1) \parallel \ldots, D\rangle \sqsubseteq_R \langle init(f_1, arg_1) \parallel \ldots, D\rangle. \tag{14}$$

  From Equation (14), we know that for any intermediate local states $(\sigma_{f_1}, \ldots)$ and database state D' during executing functions,

$$\langle \sigma_{f_1^*} \parallel \ldots, D'\rangle \sqsubseteq_R \langle \sigma_{f_1} \parallel \ldots, D'\rangle. \tag{15}$$

  According to the definition of $\sqsubseteq_R$, Equation (15) implies that every time a function instance in A_{F^*} takes one step, another function instance in A_F can always take k ($0 \leq k$) steps to result in the same database state and the same event (if any).

- A_{F^*} retries an instance. Then A_F takes no steps such that it produces the same database state and no event. The proof is similar to the second case.

  Then, the conclusion is true. □

  Finally, we can prove Theorem 1 in the paper.

Theorem 5 Given a function set F, if every function $f \in F$ satisfies idempotence simulation, then F satisfies idempotence consistency.

Proof

The premise is that every function $f \in F$ satisfies idempotence simulation.

$$\begin{aligned} &\forall f \in F. \\ &\quad (\forall D, arg. \langle init(f^*, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle). \end{aligned} \tag{16}$$

The conclusion is that F satisfies idempotence consistency (Definition 1).

From Lemma 2 and Equation (16), we imply that the execution of F can simulate the execution of F^*. That means every time A_{F^*} takes one step, A_F can take n ($n \geq 0$) steps such that they reach the same database state and produce the same event (if any). Therefore, if some execution of A_{F^*} can result in the client-observable behavior $\langle H, D\rangle$, then there exists another execution of A_F that can also result in $\langle H, D\rangle$. The conclusion is proved. □

Appendix B Proof of Failure Reduction

In this section, we prove Theorem 4 in the paper, which proves the second condition in Theorem 3. The first condition in Theorem 3 is intuitive. Thus, we omit its formal proof in this section. We first prove two lemmas and then prove Theorem 4 based on them.

Lemma 3 If the execution of f can simulate f^1, then for any $n \geq 1$, the execution of f^{n-1} can simulate f^n. The definition of f^1, f^n, and f^{n-1} are in Section 4.4.

$$(\forall D, arg. \langle init(f^1, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle) \rightarrow$$
$$(\forall D, arg, n \geq 1. \langle init(f^n, arg), D\rangle \sqsubseteq_R \langle init(f^{n-1}, arg), D\rangle).$$

Proof

The premise is

$$\forall D, arg. \langle init(f^1, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle. \tag{17}$$

We want to prove for all $n \geq 1$,

$$\forall D, arg. \langle init(f^n, arg), D\rangle \sqsubseteq_R \langle init(f^{n-1}, arg), D\rangle. \tag{18}$$

We prove it by induction on n.

Base case: When $n = 1$, Equation (18) is true, because it is equivalent to the premise Equation (17).

Inductive step: Suppose Equation (18) is true when $n = k$ ($k \geq 1$).

$$\forall D, arg. \langle init(f^k, arg), D\rangle \sqsubseteq_R \langle init(f^{k-1}, arg), D\rangle. \tag{19}$$

Then when $n = k + 1$, we want to prove

$$\forall D, arg. \langle init(f^{k+1}, arg), D\rangle \sqsubseteq_R \langle init(f^k, arg), D\rangle. \tag{20}$$

According to the definition of $\sqsubseteq_R$, we need to map every single step during the execution of f^{k+1} to s ($s \geq 0$) steps during the execution of f^k. We can construct the mapping in the following way. f^{k+1} and f^k are almost the same, except that the platform retries them for different times. Then before the first retry, we map every single step when executing f^{k+1} to a single step of f^k. That means f^{k+1} and f^k always execute the same statement. This mapping can satisfy the requirements of $\sqsubseteq_R$.

When the first retry of f^{k+1} happens, we ask f^k to be also retried. Assume the database state immediately before the retry is D_1. The executions of f^{k+1} and f^k from D_1 after the first retry are equivalent to the executions of f^k and f^{k-1} from D_1 before the first retry, respectively. This is because after the first retry, the platform will retry f^{k+1} for k times and retry f^k for $k-1$ times. From Equation (19), we know that

$$\forall arg. \langle init(f^k, arg), D_1\rangle \sqsubseteq_R \langle init(f^{k-1}, arg), D_1\rangle. \tag{21}$$

Then there exists a step mapping from every step of f^{k+1} to steps of f^k after the first retry, which satisfies the requirements of $\sqsubseteq_R$. Therefore, Equation (18) is true for $n = k + 1$.

Conclusion: By the principle of induction, Equation (18) is true for any $n \geq 1$. □

Lemma 4 For any $i, j, k \geq 0$, if the execution of f^k can simulate f^j and the execution of f^j can simulate f^i, then the execution of f^k can simulate f^i.

$$\forall i, j, k.$$
$$((\forall D, arg. \langle init(f^i, arg), D\rangle \sqsubseteq_R \langle init(f^j, arg), D\rangle) \wedge$$
$$(\forall D, arg. \langle init(f^j, arg), D\rangle \sqsubseteq_R \langle init(f^k, arg), D\rangle))$$
$$\rightarrow (\forall D, arg. \langle init(f^i, arg), D\rangle \sqsubseteq_R \langle init(f^k, arg), D\rangle).$$

Proof

This lemma is similar to the transitivity of forward simulation. The premises include

$$\forall D, arg. \langle init(f^i, arg), D\rangle \sqsubseteq_R \langle init(f^j, arg), D\rangle, \tag{22}$$

and

$$\forall D, arg. \langle init(f^j, arg), D\rangle \sqsubseteq_R \langle init(f^k, arg), D\rangle. \tag{23}$$

The conclusion is

$$\forall D, arg. \langle init(f^i, arg), D\rangle \sqsubseteq_R \langle init(f^k, arg), D\rangle. \tag{24}$$

We use σ_i, σ_j, and σ_k to denote the local state when executing f^i, f^j, and f^k, respectively. We will prove the conclusion Equation (24) by *co-induction*. According to the definition of $\sqsubseteq_R$, every step taken during executing f^i belongs to one of the following three cases.

- $\langle \sigma_i, D\rangle \xrightarrow{\alpha} \langle \sigma_i', D'\rangle$, where α is not a *response* event.

  According to the definition of $\sqsubseteq_R$, we need to prove that there exists σ_k' such that $\langle \sigma_k, D\rangle \overset{\alpha}{\Longrightarrow} \langle \sigma_k', D'\rangle$ and $\langle \sigma_i', D'\rangle \sqsubseteq_R \langle \sigma_k', D'\rangle$.

  From Equation (22), we know that there exists σ_j' such that

  $$\langle \sigma_j, D\rangle \overset{\alpha}{\Longrightarrow} \langle \sigma_j', D'\rangle, \tag{25}$$

  and

  $$\langle \sigma_i', D'\rangle \sqsubseteq_R \langle \sigma_j', D'\rangle. \tag{26}$$

  From Equation (23) and Equation (25), we know there exists σ_k' such that

  $$\langle \sigma_k, D\rangle \overset{\alpha}{\Longrightarrow} \langle \sigma_k', D'\rangle, \tag{27}$$

  and

  $$\langle \sigma_j', D'\rangle \sqsubseteq_R \langle \sigma_k', D'\rangle. \tag{28}$$

  From Equation (26) and Equation (28), we know that

  $$\langle \sigma_i', D'\rangle \sqsubseteq_R \langle \sigma_k', D'\rangle. \tag{29}$$

- $\langle \sigma_i, D\rangle \xrightarrow{\alpha} \langle \sigma_i', D'\rangle$, where α is a *response* event.

  According to the definition of $\sqsubseteq_R$, we need to prove that there exists σ_k' such that $\langle \sigma_k, D\rangle \xRightarrow{\alpha} \langle \sigma_k', D'\rangle$.

  From Equation (22), we know that there exists σ_j' such that

$$\langle \sigma_j, D\rangle \xRightarrow{\alpha} \langle \sigma_j', D'\rangle. \tag{30}$$

  From Equation (23) and Equation (30), we know there exists σ_k' such that

$$\langle \sigma_k, D\rangle \xRightarrow{\alpha} \langle \sigma_k', D'\rangle. \tag{31}$$

- $\langle D, D'\rangle \in R$, which means that other concurrent functions change the database state from D to D'.

  According to the definition of $\sqsubseteq_R$, we need to prove that $\langle \sigma_i, D'\rangle \sqsubseteq_R \langle \sigma_k, D'\rangle$.

  From Equation (22), we know

$$\langle \sigma_i, D'\rangle \sqsubseteq_R \langle \sigma_j, D'\rangle. \tag{32}$$

  From Equation (23), we know

$$\langle \sigma_j, D'\rangle \sqsubseteq_R \langle \sigma_k, D'\rangle. \tag{33}$$

  From Equation (32) and Equation (33), we know

$$\langle \sigma_i, D'\rangle \sqsubseteq_R \langle \sigma_k, D'\rangle. \tag{34}$$

  Thus, we have proved the conclusion Equation (24). □

Finally, we can prove Theorem 4 in the paper based on the above two lemmas.

Theorem 6 Given a function f, if each execution with one retry under concurrency has a corresponding retry-free execution that can simulate it, then each execution with arbitrary times of retries also has a corresponding retry-free execution that can simulate it.

$$(\forall D, arg. \langle init(f^1, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle) \rightarrow$$
$$(\forall D, arg, n \geq 1. \langle init(f^n, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle).$$

Proof

The premise is

$$\forall D, arg. \langle init(f^1, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle. \tag{35}$$

The conclusion is

$$\forall D, arg, n \geq 1. \langle init(f^n, arg), D\rangle \sqsubseteq_R \langle init(f, arg), D\rangle. \tag{36}$$

From Lemma 3 and the premise Equation (35), we know

$$\forall D, arg, n \geq 1. \langle init(f^n, arg), D\rangle \sqsubseteq_R \langle init(f^{n-1}, arg), D\rangle. \tag{37}$$

From Lemma 4 and Equation (37), we know

$$\forall D, arg, n \geq 1. \langle init(f^n, arg), D\rangle \sqsubseteq_R \langle init(f^0, arg), D\rangle. \tag{38}$$

Since f^0 is equivalent to f, Equation (38) is equivalent to the conclusion Equation (36). The conclusion is true. □

```
1 int f(input)
2 {
3   output = f1(input);
4   return f2(output);
5 }
```

Figure 12: A function composed of two sub-functions called f_1 and f_2.

Appendix C Proof of Theorem 2

This section proves Theorem 2 in Section 4.3 of the paper. We first define and prove sequential compositionality of idempotence simulation.

Definition 2 Given a function f and the rely condition R, f satisfies strong idempotence simulation means that: 1) f satisfies idempotence simulation under R; and 2) after the platform successfully executes f without retries for one time, retrying f again will not modify the shared state.

Particularly, the second condition is equivalent to the third requirement in Theorem 2 of paper: it will not affect the shared state on retry once it has been successfully executed.

Lemma 5 Given any two functions f_1 and f_2, if f_1 satisfies strong idempotence simulation, f_2 satisfies idempotence simulation, and the input of f_2 remains unchanged on retry, then the function f in Figure 12 composed of f_1 and f_2 also satisfies idempotence simulation.

Proof According to Theorem 3, we prove the simulation relation between f^1 and f. Note that f^1 is defined in Section 4.4, which is different from f_1. Then we classify the location where retry occurs during the execution of f^1 into three cases and prove that under all these cases, executing f without retries could exhibit all possible client-observable behaviors produced by executing f^1. Then we can prove that f satisfies idempotence simulation.

- Retry happens during the execution of f_1. Then the execution of f^1 consists of three main parts:
  (P1) the execution of f_1 before retry;
  (P2) the normal execution of f_1 after retry;
  (P3) the normal execution of f_2.
  Since f_1 satisfies idempotence simulation, a normal execution of f_1 without retries could exhibit all client-observable behaviors of (P1) and (P2). In this case, a normal execution of f without retries can simulate the execution of f^1.

- Retry happens between f_1 and f_2. This execution exhibits the same client-observable behavior as another execution where retry happens immediately before the "return" statement in f_1. The proof is the same as the first case.

- Retry happens during the execution of f_2. Then the execution of f^1 consists of four main parts:
  (P1) the first normal execution of f_1 without retries;

```
1 int f(int input)
2 {
3   int output1 = f1(input);
4   int output2 = f2(output1);
5   ...;
6   return fn(outputn);
7 }
```

Figure 13: A function composed of n sub-functions called f_1, ..., and f_n.

(P2) the execution of f_2 before retry;
(P3) the second normal execution of f_1 without retries;
(P4) the normal execution of f_2 without retries.
Since f_1 satisfies strong idempotence simulation, the second normal execution of f_1 will not modify the shared state and return the same value as the input for the f_2. Therefore, this kind of execution of f^1 exhibits the same client-observable behavior as the execution composed of (P1), (P2), and (P4). Since f_2 satisfies idempotence simulation, there exists another normal execution of f_2 that exhibits the same client-observable behavior as the execution of (P2) and (P4). Thus, the execution of f^1 described in this case exhibits the same client-observable behavior as another normal execution of f without retries.

In conclusion, for any execution of f^1, there always exists a normal execution of f that exhibits the same client-observable behavior under concurrency. According to Theorem 4, f satisfies idempotence simulation. □

Then we extend sequential compositionality to a function composed of arbitrary number of code fragments.

Lemma 6 For any positive integer $n \geq 2$ and any function f in the form of Figure 13, if each f_i $(1 \leq i < n)$ satisfies strong idempotence simulation, f_n satisfies idempotence simulation, and the input of each f_i remains unchanged on retry, then f satisfies idempotence simulation.

Proof We prove it by induction on n ($n \geq 2$). The premise is Lemma 5.

Base case. When $n = 2$, the conclusion is true, because it is equivalent to the premise.

Inductive step. Assume the conclusion is true when $n = k$. We prove that the conclusion is also true when $n = (k+1)$. We can treat the code fragment containing the first k sub-functions as a function g. Since the conclusion is true when $n = k$, the function g satisfies idempotence simulation. f consists of two sub-functions called g and f_{k+1}, both of which satisfy idempotence simulation. From Lemma 5, we can imply that f satisfies idempotence simulation.

Conclusion. By the principle of induction, f satisfies idempotence simulation for any $n \geq 2$. □

Based on Lemma 6, we prove that our method of addressing unbounded loops with write operations is sound, which is

```
1  void checkCoupons(coupons, time) {
2    // C1
3    if(coupons.size == 0)
4      return;
5    // L
6    for(int j = 0; j < coupons.size(); j++) {
7      coupon := get("Coupon", coupons[j].couponId);
8      if(isExpired(coupon.date, time)) {
9        coupon.expired := true;
10       put("Coupon", coupons[j].couponId, coupon);
11     }
12   }
13   // C2
14   return;
15 }
```

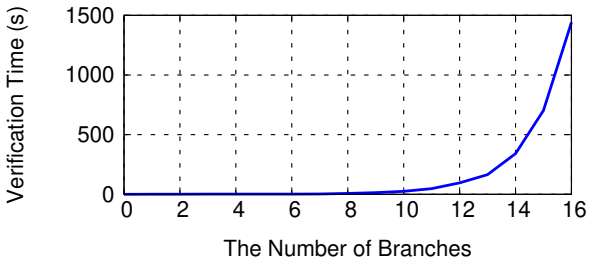
Figure 14: An example of unbounded loop with write operations.

Theorem 2 in the paper. For convenience, we represent a function with an unbounded loop as $\{C_1; L; C_2\}$, where L is the unbounded loop, C_1 is all code preceding L, and C_2 denotes all code following the loop. B_L is the loop body of L.

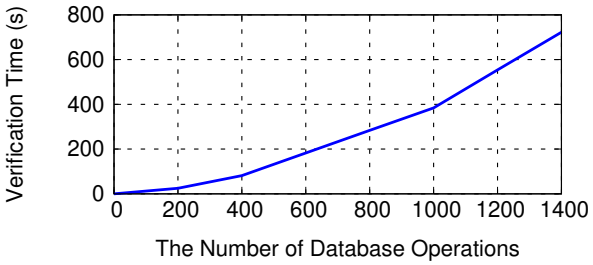
Theorem 7 Given a function f with the unbounded loop in case 2, f satisfies idempotence simulation if the number of iterations of the loop L remains unchanged on retry, and C_1, C_2 and B_L can satisfy the following requirements: 1) They all satisfy idempotence simulation; 2) Their inputs do not change on retry; 3) They will not affect the shared state on retry once the function has successfully executed them.

Proof Since the number of loop iterations is the same on retry, the execution of f comprises the execution of C_1, the execution of an unbounded number of B_L, and the execution of C_2. We can prove that f satisfies idempotence simulation based on Lemma 6. Although the number of loop iterations is unbounded, the loop body executed by each iteration is the same. Therefore, we just need to prove that C_1, B_L, and C_2 satisfy the requirements in Lemma 6. The requirements have been ensured by the premise of Theorem 7. Therefore, the conclusion is true and f satisfies idempotence simulation. □

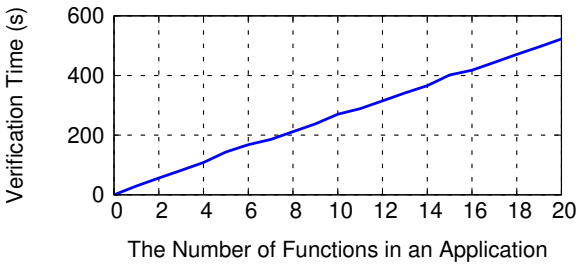
We use an example to show how to use this theorem to perform the verification. In Figure 14, the *checkCoupons* function uses an unbounded loop to check whether coupons have expired. Obviously, C_1 and C_2 in *checkCoupons* satisfy all requirements in Theorem 2. The number of the loop iterations is the size of *coupons* which will be the same on retry. Thus, to prove the idempotence simulation of *checkCoupons*, we only need to focus on the loop body B_L (line 7-11). First, Flux can prove that B_L satisfies the idempotence simulation; Then, it uses static analysis to find that B_L's input is *coupons* that keeps consistent on retry. Third, once it successfully updates *expire* to be *true* for a specific *coupon*, the value will remain unchanged as *checkCoupons* always updates it to be *true* on retry. According to the theorem, we have that *checkCoupons* satisfies the idempotence simulation.



(a) The verification time of a single function with ten database operations and different numbers of branches.



(b) The verification time of a single function with different numbers of database operations. The function has one execution path. Lines of code also increase linearly when the number of database operations increases linearly.



(c) The verification time of an application with different numbers of functions.

Figure 15: The verification time for different numbers of branch statements, database operations, and functions.

Appendix D Scalability of the Verifier

We have created micro-benchmarks to evaluate the scalability of the verifier. Figure 15a shows that when the number of branches in a single function increases, the verification time increases exponentially, as the number of traces also increases exponentially. Besides, Figure 15b shows that when the number of database operations in a single function increases, the verification time increases linearly. This is because the number of generated traces increases linearly. Note that because these micro-benchmarks mainly contain database operations, LoC also increases linearly when the number of database operations increases. Thus, Figure 15b also demonstrates that when the LoC of a single function increases, the verification time increases linearly. Additionally, we evaluate the verification time for an application with different numbers of functions. Each function has one execution path and two hundred database operations. Figure 15c shows that the verification time increases linearly when the number of functions in an application increases linearly. Because the number of traces increases linearly.