SemantiX: A Compatibility Checker between Applications and Compositions of Distributed Systems

YIFEI SUN, Northeastern University, USA
JI-YONG SHIN, Northeastern University, USA

Modern distributed applications compose multiple services with different consistency guarantees. Mismatches between application requirements and chosen system semantics can introduce subtle bugs, but verifying compatibility across complex applications is both challenging and time-consuming. In particular, existing testing approaches or rigorous formal verification lack flexibility and cannot promptly provide correctness guarantees on multi-semantic compositions.

We present SemantiX, a framework for checking semantic compatibility between applications and compositions of distributed systems with heterogeneous consistency models. SemantiX embeds formally defined distributed system modules of consistency semantics which include the first formal definition of visibility constraints. SemantiX introduces the AppGraph approach for systematically modeling complex applications. Applications modeled using AppGraphs can be checked for their compatibility against a combination of distributed system modules that the user is considering. Alternatively, SemantiX can search for combinations of modules compatible with the application. We present three case studies on a movie streaming service, an e-commerce platform, and a cross-service causal distributed storage service. We demonstrate that SemantiX can capture complex service compositions with minimal encoding effort and quickly check compositional compatibility with underlying systems or find the compatible system configurations. SemantiX enables developers to efficiently verify distributed application designs and explore alternative consistency configurations, supporting more agile development.

ACM Reference Format:
Yifei Sun and Ji-Yong Shin. 2026. SemantiX: A Compatibility Checker between Applications and Compositions of Distributed Systems. *Proc. ACM Softw. Eng.* , ISSTA (October 2026), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern cloud environments provide hundreds of services on various platforms that are readily available for developers to build a large-scale distributed application. System architects and developers following the microservice or service-oriented designs can freely choose and compose the building blocks in a modular fashion and plug them into the application.

However, different distributed storage systems provide different consistency semantics, and the mismatch of the chosen system's semantics and the application's expected semantics could introduce subtle and silent semantic bugs [32, 33]. For example, a user may connect a key-value store providing eventual consistency to an application that expects a linearizable storage service. In this case, the application could read outdated information causing erroneous outcomes but without crash-like explicit errors.

Authors' Contact Information: Yifei Sun, ysun@ccs.neu.edu, Northeastern University, Boston, Massachusetts, USA; Ji-Yong Shin, j.shin@northeastern.edu, Northeastern University, Boston, Massachusetts, USA.

ACM 2994-970X/2026/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Developers and system architects could carefully review the application needs and choose a matching system. However, when the application depends on multiple distributed services, checking the overall compatibility between the application logic and multiple system components can easily become unmanageable. Even if one successfully finds the correct combination, for example, through extensive testing, developers might change the application code or plug in different systems due to management, considerations of cost and performance. Consequently, sophisticated compatibility checks must be repeated.

Formal verification offers an alternative approach to these challenges, where the correctness of the code can mathematically guarantee the compatibility between code and system [26]. However, depending on the complexity of the code, which could include non-essential parts for reasoning about compatibility, the mechanized formal proofs can take extensive time and effort [19, 23, 39]. Furthermore, depending on the technique used, mechanized formal proofs can be several times larger, in Lines of Code (LoC), than the implementation. Moreover, minor changes can easily invalidate the entire proof [23]. Thus, formal verification leads to a similar problem to above-mentioned testing methods where the compatibility check lacks the agility to adapt to changes in underlying systems and applications.

In this paper, we propose SemantiX, a lightweight semantic compatibility checking framework between applications and composition of distributed systems. SemantiX helps developers quickly encode their application logic and requirements and check the compatibility with underlying distributed systems using an SMT-based approach.

SemantiX includes axiomatic models of distributed consistency semantics, including eventual consistency, linearizability [48], four session guarantees [43], pipeline RAM (PRAM) consistency [30], and causal consistency [3], which are commonly used in both industry and academia [1, 2, 11, 18, 31, 38, 44, 45, 50]. Users of our tool can leverage predefined models as building blocks to rapidly iterate the service compositions and compatibility of these models with their applications. Additional models can be added to further strengthen SemantiX's expressiveness with provided interfaces. User-authored custom models inheriting from our base class automatically work with our pre-built ones, including assertions for individual satisfiability and compatibility between one or more models with user defined application logic.

Although axiomatic definitions of distributed consistency semantics exist in prior literature [14, 16, 24, 34, 41], encoding the details into the SMT clauses for compatibility checks is challenging as the definitions omit key details. For example, the commonly used concept of *visibility* for defining weak distributed consistency semantics is subtle, but to our knowledge, no literature provides a concrete definition. SemantiX fills in these missing gaps to realize full-fledged formal models for the compatibility checking.

Each application functionality is systematically encoded as a directed acyclic graph (DAG), and the entire application is modeled as a directed multi-graph format which we call the AppGraph. Each node in the AppGraph represents key functionality in the application that requires certain properties from the lower-level nodes it depends on, and provides certain properties to the upper-level nodes. The edges are used to specify more specific requirements and condition that each functionality needs to enforce in accessing the downstream nodes. The leaf nodes typically represent external distributed services following one of the pre-built distributed consistency semantic models in SemantiX. The AppGraph model is expressive enough to model an application logic from high-level and trace complex dependencies that go across multiple application logic boundaries, and distributed systems which the application relies on.

Using the AppGraph, SemantiX can verify simple applications relying on one distributed service and scale to complex microservice-based production environment that leverages multiple modules and distributed storage systems. Once the user specifies the application logic, SemantiX can quickly

check the compatibility with the plugged-in consistency models. This could be verifying the user's selection, or alternatively, SemantiX can explore different combinations of existing models to find compositions of distributed system with compatible consistency semantics. These approaches enable fast exploration and iteration of alternative system designs, guiding the development to adhere to intended consistency guarantees from the very beginning.

As case studies, we use SemantiX to model two microservice-based applications [17], a movie streaming service, an e-commerce service, and a cross-service causal storage system built on top of multiple weakly consistent distributed storage systems [4]. We evaluate and demonstrate SemantiX's ability to check compatibility between user-defined application specific logic, with underlying distributed systems and explore other compatible semantics. Our framework is capable of checking single compatibility combination in under 20 milliseconds and exploring other compatible combinations in less than 5 minutes.

This paper makes the following contributions:

- We present SemantiX, the first compatibility checking tool for applications that are composed of multiple distributed services.
- We provide the missing formal definitions including visibility to fully express and check distributed weak consistency semantics.
- We present a systematic way of encoding complex applications using the AppGraph structure and checking their compatibility collectively using the SMT-based approach.
- We conduct case studies on three representative distributed applications to demonstrate the workings of SemantiX.

2 Background and Motivating Example

2.1 Background

Distributed consistency semantics define in which order updates are applied and made visible in replica nodes within a cluster of distributed system. When an update request is submitted to the distributed system, a subset (typically one) of nodes receives it and the update is propagated across distributed nodes. Due to the asynchronous nature of the network, replica nodes receive the update with different delays and users accessing distinct nodes could observe different data states. Distributed consistency semantics introduce ordering constraints on top of such non-determinism so that applications can predictably communicate with the system.

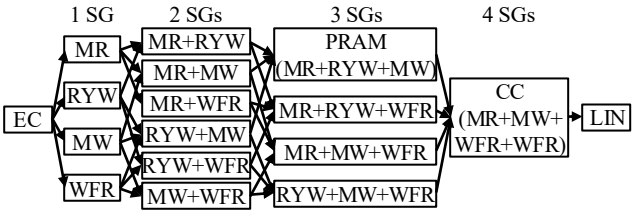

Fig. 1. Partially ordered semantics. Strictly stronger ones (destinations of the arrows) subsume the strictly weaker ones (sources of the arrows).

Among many distributed consistency semantics, SemantiX models seventeen widely used semantics in systems from both industry and academia [1, 2, 11, 18, 31, 38, 44, 45, 50], which includes eventual consistency [9], session guarantees [43], causal consistency [3], and linearizability [21].

While eventual consistency (EC) is considered a liveness property, it defines a minimum safety property that visible data reflects valid updates in history. Informally,

- **Eventual consistency (EC):** users may read any versions of data but with no incoming updates the most up-to-date version will eventually become visible to all reads.

Session guarantees (SGs), like its name, define semantics from the viewpoint of a user session:

- **Monotonic reads (MR):** a user sees the same or later version of data than what was last seen by the same user.
- **Read-your-writes (RYW):** a user sees the same or later version of data than what was last written by the same user.
- **Monotonic writes (MW):** writes by a user apply in the order that this user wrote.
- **Write-follows-reads (WFR):** writes by a user apply after writes corresponding to what this user read apply.

By choosing whether to enforce each of four session guarantees, a total of 16 (= 2^4) different semantics can be derived. While most of them do not have specific names (e.g., MR + RYW), there are a few well-known named semantics. For example, EC does not enforce any of SGs, pipeline RAM consistency (PRAM) is a combination of MR, RYW, and MW [8], and causal consistency (CC) is a combination of all four SGs, MR, RYW, MW, and WFR [7]. Still, PRAM [30] and CC [3] have their own definitions:

- **PRAM consistency (PRAM):** read and write operations by a user happen in first in first out order from this user's point of view, but there are no global ordering guarantees.
- **Causal consistency (CC):** read and write operations follow the causal ordering (or happens-before ordering [25]) .

Finally, linearizability hides all distributed and concurrent natures of distributed systems.

- **Linearizability (LIN):** all read and write operations are applied as if they are executed atomically by a single non-concurrent user.

These semantics can be partially ordered by their strength [48] where strictly stronger semantics subsumes the weaker ones (Figure 1). Thus, when an application requires a certain semantics then a strictly stronger one can safely replace the strictly weaker one without consistency violations.

2.2 Motivating Example

Assume a movie streaming service built using a form of microservice architecture where admins manage the serviced content and clients rent movies and leave reviews [17] (Figure 5). There are quite a few functionalities and components to build the service, and it is nontrivial to meet all service requirements and to guarantee that all requirements are fulfilled.

For example, a movie metadata database (DB) stores list of available movies and each movie's information such as the logline, director, and cast. This DB is accessed 1) when the service admin adds/modifies movie information and 2) when clients read the DB before renting a movie or writing a movie review.

The developer of the system can choose an ACID database or linearizable key-value store for the metadata DB, but this may not scale well as stronger consistency models trade stronger data consistency guarantee with higher latency. For utmost scalability and performance, an eventually consistent key-value store can be used, but users may observe anomalies: for example, reviews written to one replica may not be visible to the same user reading it. Thus, to find a proper storage module, the key constraints are to identify each access functionality, and to minimize anomalies users may experience. For example, we can define admins must always observe their own updates and clients must be able to see what they have previously seen. These constraints must be met by the consistency model of the metadata DB that the application developer chooses to use. However, as functionalities of the system grow, constraints typically become more sophisticated and the complexity of checking whether the constraints are fully satisfied by the DB also grows accordingly.

While the metadata DB example above is presented in isolation from what a single DB service must satisfy, cross-DB constraints can exacerbate the problem. For example, accessing the metadata DB would require checking whether the user is an admin or a client with active movie subscription status which may be maintained in a separate storage system. Thus, dependencies from the login

service and the user DB must be met: the user must be able to observe the user's valid up-to-date status and then access the other services. Renting a movie or reading/writing a review involves additional consistency constraints that cross storage services for movies and reviews forming more complex dependencies that the system as a whole must satisfy.

SemantiX provides facilities to systematically encode the constraints and check whether the constraints are all satisfied.

3 Distributed Consistency Models in SemantiX

First, we present the theory and formal details regarding how we modeled the distributed consistency semantics, and how applications can be encoded into our SemantiX framework, then further discharged to SMT solvers like Z3 [12] or CVC5 [6].

3.1 Preliminaries

SemantiX provides pre-built and pluggable distributed system modules (DMs) that represent distributed storage services, which can be checked individually or in group, or as application constraints. Each DM models distributed consistency semantics introduced in Section 2 and Figure 1. In the movie streaming service example, given all the constraints for the metadata DB and the developer's DM choice for the metadata DB, SemantiX can check if they are compatible. Also, given the constraints, SemantiX can search existing DMs for compatible consistency semantics.

SemantiX follows the definitions provided by Viotti et al. [48] and others [5, 9, 43] and uses predicate logic derived from the definitions to model DMs and reason about consistency semantics with SMT solvers. Consistency semantics are defined as logical formulas over axioms and relations. Each model enforces safety properties, specifically ordering and visibility constraints on operations within history and abstract executions.

Operations, Relations, and History. To model a consistency semantics for a DM, we must first define an operation, a tuple containing the following:

- *proc:* the process that issued the operation which is used for session identification.
- *type:* the type of the operation such as read, write, or custom.
- *obj:* the object the operation is performed on.
- *ival:* operation's input value.
- *oval:* operation's output value or a special marker used for no output value.
- *stime:* invocation time of the operation.
- *rtime:* return time of the operation or a special marker used for non-returning operations.

An operation can be conceptualized as an atomic action with a request-response pair, where the request is the operation itself and the response is the other operation with the response time set.

Consistency semantics are typically defined using universally quantified formulas (e.g., $\forall a, b : \ldots$) applied to relations that are treated as uninterpreted functions with constraints of operations by the SMT solver (e.g., $\text{vis}(a, b)$, $\text{so}(a, b)$). Typically, they are pairwise constraints defined over operations in history. A history is a set of operations that have been logically executed in the DM. The functions representing the relations are used as base premises for consistency models and are used in axioms and invariants that the SMT solvers interpret and check. Given that we extensively use set comprehension to capture relations between operations, throughout the formalization, $a \xrightarrow{\text{rel}} b$ is denotationally equal to $(a, b) \in rel$, and directly translate to `rel(a, b)` in SemantiX's source code. Further, rel^{-1} is equivalent to the inverse of rel relation [9, 48].

The relations defined for operations within a history include:

- returns-before: $\text{rb} \triangleq \{(a, b) : a, b \in H \wedge a.\text{rtime} < b.\text{stime}\}$
- same-session: $\text{ss} \triangleq \{(a, b) : a, b \in H \wedge a.\text{proc} = b.\text{proc}\}$

- session-order: so $\triangleq$ rb $\cap$ ss
- same-object: ob $\triangleq \{(a, b) : a, b \in H \wedge a.\text{obj} = b.\text{obj}\}$

In our implementation, each history relation function accepts an optional `symbols` argument. This parameter allows users to explicitly specify the symbolic operations, for example, `["a_X", "b_X"]`, over which the relation's constraints should be quantified for that specific instance `X`. This mechanism enables the application of history-based constraints distinctly to operations belonging to different modeled systems or components, contributing to the logical isolation necessary for cross-system analysis.

Abstract Execution (AE). An abstract execution instantiates and refines the history by specifying which operations are visible to, or are arbitrated over each other to model and check valid behaviors under a consistency semantics. Abstract execution relations are non-deterministic orderings used to capture more complex temporal constraints and conflict resolution. The relationships include:

- *Visibility (vis):* defines the effects of write propagation where the effect of an operation is visible to the process invoking another operation.
- *Arbitration (ar):* an application-specific, transitive and acyclic relation that provides a total order on conflicting operations, ensuring that observed executions follow a single coherent timeline.
- *Happens-before (hb):* used to capture probable causality, and is previously defined in the literature as the transitive union of session order and visibility: $hb \triangleq (\text{so} \cup \text{vis})^+$

The formal definitions of these relationships can be found in prior work [9, 48] with the exception of visibility which we define in Section 3.3. These relations provide the foundation for modeling sophisticated consistency guarantees that go beyond simple temporal ordering, enabling the specification of complex distributed system behaviors where operations may be reordered, delayed, or selectively visible across different system components. In other words, multiple AEs are possible for a single history as observed event orderings can differ between nodes. The AE encode the non-deterministic effects of asynchronous execution environments and implementation-specific constraints.

3.2 Base Consistency Semantics

Building on these preliminaries, we now discuss the formal definitions of base consistency semantics: EC, MR, RYW, MW, WFR, PRAM, CC, and LIN. In Section 2., the semantics are defined axiomatically using the relations between pairwise operations (i.e., a, b, and c in a history H of operations, where H_{rd} and H_{wr} are histories of reads and writes respectively. While we adopt the definitions of MR, RYW, MW, WFR, and PRAM of Viotti et al. [48] (Figure 2), we introduce or augment EC, CC, and LIN for SMT encoding and check their correctness with respect to other semantic definitions using SemantiX (Section 6).

$$
\begin{aligned}
\text{MR} &\triangleq \forall a \in H, \forall b, c \in H_{\text{rd}} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{vis}} c \\
\text{RYW} &\triangleq \forall a \in H_{\text{wr}}, \forall b \in H_{\text{rd}} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{vis}} b \\
\text{MW} &\triangleq \forall a, b \in H_{\text{wr}} : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{ar}} b \\
\text{WFR} &\triangleq \forall a, c \in H_{\text{wr}}, \forall b \in H_{\text{rd}} : a \xrightarrow{\text{vis}} b \wedge b \xrightarrow{\text{so}} c \Rightarrow a \xrightarrow{\text{ar}} c \\
\text{PRAM} &\triangleq \forall a, b \in H : a \xrightarrow{\text{so}} b \Rightarrow a \xrightarrow{\text{vis}} b
\end{aligned}
$$

Fig. 2. Formal definitions of monotonic reads (MR), read-your-writes (RYW), monotonic writes (MW), write-follows-reads (WFR), and pipeline RAM (PRAM) consistency.

Eventual Consistency (EC). While EC definitions in other literature [43, 48, 49] focus primarily on liveness aspects using the concept of eventual visibility of operations, SemantiX excludes liveness properties and concentrates on safety properties: (1) non-circular causality, an acyclic projection of the happens-before relation, and (2) contextual return value consistency, a predicate on abstract

executions ensuring that the return value of any operation within that execution belongs to the set of its valid return values guided under the can-view relation which we define in Section 3.3.

$$\mathrm{EC}(\mathcal{F}) \triangleq \mathrm{acyclic}(hb) \wedge \forall op \in H : op.oval \in \mathcal{F}(op, ctx(A, op))$$
$$\mathcal{F}_{\text{can-view}}(op, ctx(A, op)) = \{x.ival \in H_{\text{wr}} : (op, x) \in \text{can-view}\}$$

Causal Consistency (CC). Our CC formulation combines definitions from Viotti et al. [48], Burckhardt et al [9], and causal memory [5]. The *writes-into (wi)* relation [5] connects write operations directly with the reads that retrieve their values. This guarantees that when a read operation observes a specific write, all following writes within the same session honor that causal sequence. The formalization coordinates session order, arbitration, visibility, and writes-into to preserve consistent causal histories.

Operations belong to the `wi` set when: a write a writes into a read b if and only if b retrieves the value initially written by a, and a and b access the same object (or identical memory region). Each b can have at most one corresponding a, maintaining acyclicity. When one operation succeeds another in session order, their relationship within the abstract execution faces additional constraints. Particularly, if $(a, b) \in$ so, then a must write-into b when b represents a read operation, and a must be visible and arbitrated prior to b. Consequently, session order establishes a causal sequence that manifests in the relations wi, vis, and ar.

$$\mathrm{CC} \triangleq \mathrm{WFR} \wedge \forall a, b \in H : (a, b) \in \mathrm{so} \Rightarrow (a, b) \in \mathrm{wi} \cap \mathrm{vis} \cap \mathrm{ar}$$

These constraints collectively establish a causal memory model where session order, observed values, and write sequences maintain alignment, ensuring causally dependent writes manifest in proper order from any observer's viewpoint.

Linearizability (LIN). In our encoding of LIN, we introduce a modified single global order constraint to unify visibility and arbitration for all write operations regardless of operations in session-based semantics or not, and enforce real-time ordering to ensure that the observed histories comply with returns-before relations: i.e., linearization as operations come in instead of lazily ordering events when reads occur [51].

$$\begin{aligned}\mathrm{LIN} \quad \triangleq \quad & (\forall a, b \in H : [a \xrightarrow{\text{vis}} b \implies a \xrightarrow{\text{ar}} b] \wedge [a \xrightarrow{\text{ar}} b \wedge a.rtime < \infty \implies a \xrightarrow{\text{vis}} b]) \\ \wedge \quad & rb \subseteq ar \quad \wedge \quad \forall x \in H : x.oval \in \{y.ival \in H_{\text{wr}} : (x, y) \in \text{can-view}\}\end{aligned}$$

3.3 Formal Definitions of Visibility

One of our key contributions is formally defining visibility. Although visibility is used as part of defining almost all semantics in Section 3.2 and Figure 2, its definitions in existing literature is informal and ambiguous regarding exact behavior under concurrent settings [15, 47, 48, 51]. As with many seemingly simple informal definitions concerning concurrent and distributed environments, formally defining visibility for use in SemantiX-like tools is non-trivial. We restructure visibility as a binary relation and perform case analysis on all possible combinations of read and write operations that can be *visible* to each other.

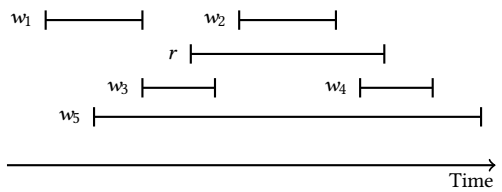

Fig. 3. The can-view relation captures the possible visibility between read and write operations. The read-write pairs r and w_{1-5} are defined based on their start and return times, where the read can view the effect of the write operations.

We introduce the notions of *can-view* and *viewed* to capture the deterministic and non-deterministic behavior of operations, and what it means for one operation to be able to *observe* the effects of another operation. To achieve visibility, two operations must first fall in one of the categories in *can-view*: either (1) the write returns before read (*e.g.*, w_1 and r in Figure 3; note that which process issued the read/write does not matter, as synchronization is not required to establish probable effect causality) or (2) the read and write are *concurrent* (*i.e.*, returns before relations do not hold as the durations of the read and write overlap as in r and w_{2-5} in Figure 3) such that the read is eligible to observe the write's effect. Formally, this can-view relation, $a \xrightarrow{\text{can-view}} b$, is defined over the pairwise start or return time of read a and write b operations:

$$\text{can-view} \triangleq \{(a,b) : a \in H_{\text{rd}}, b \in H_{\text{wr}} \wedge a.\text{obj} = b.\text{obj} \quad \wedge \quad (b \xrightarrow{\text{rb}} a \vee \neg(a \xrightarrow{\text{rb}} b \vee b \xrightarrow{\text{rb}} a))\}.$$

While can-view captures the possible visibility between read and write, the result dependency between them is captured by the *viewed* relation:

$$\begin{aligned} \text{viewed} \quad \triangleq \quad & \{(a,b) : a \in H_{\text{rd}}, b \in H_{\text{wr}} \quad \wedge \quad (a,b) \in \text{can-view} \\ & \wedge((a.\text{oval} = b.\text{ival}) \quad \vee \quad (\exists x \in H_{\text{wr}} : (x,b) \in \text{can-view} \wedge a.\text{oval} = x.\text{ival}))\} \end{aligned}$$

The viewed relation is defined as a non-deterministic pairwise partial ordering between a read and a write that builds upon can-view. Aside from timestamps falling into one of the can-view cases, we assign additional value-related constraints to operations: either the value written by the write operation must match the output of the read, or, in case of another write operation that happened after or concurrently to the aforementioned write, the viewed relation enforces the output of the read to be from one of the writes (but only one can be chosen). In the visibility definition, the transitivity of the viewed relation is implicitly enforced.

Using Figure 3's operations r and w_1 as an example, since r and w_1 are in the can-view relation, if w_1's written value matches r's output, then r is in the viewed relation with w_1. Consider another operation w_5 in the same figure that happened concurrently with w_1. If w_5's written value differs from w_1, but r and w_5 are in the can-view relation and r and w_5 have value equivalence, then r and w_1 are still in the viewed relation (write w_1 still affects r). However, in this scenario, the constraint in effect for the viewed relation is the existence of w_5 and the value equivalence of r and w_5.

Finally, the *visibility (vis)* definition establishes a deterministic pairwise relationship between any two operations in the history rather than being limited to read-write pairs. It also adds the transitivity (propagation) and acyclicity (cannot go back in time) to the viewed relation. These were not enforced in our original viewed definition to preserve non-determinism in ordering concurrent operations and to include all pairs of operations that can be viewed by each other. When constraint $a \xrightarrow{\text{vis}} b$ is assigned to a pair of operations, the following four cases are considered:

$$\begin{aligned} \text{vis} \triangleq \{(a,b) : & (a \in H_{\text{rd}}, b \in H_{\text{rd}} : \exists x \in H_{\text{wr}} : (x,a) \in \text{vis} \Rightarrow (x,b) \in \text{vis}) \\ & \vee (a \in H_{\text{rd}}, b \in H_{\text{wr}} : \exists x \in H_{\text{wr}} : (x,a) \in \text{vis} \Rightarrow (x,b) \in \text{vis}) \\ & \vee (a \in H_{\text{wr}}, b \in H_{\text{rd}} : \text{viewed} \subseteq \text{ar}) \quad \vee \quad (a \in H_{\text{wr}}, b \in H_{\text{wr}} : (a,b) \in \text{ar})\}. \end{aligned}$$

Note that the read-read and read-write cases are recursively defined, and we call this the write propagation rule. Conceptually, the rule implies that that, if a read a and another operation (either read or write) b are in the visibility relation, the closest write operation x that is visible to read a must be propagated to b. Whether this visibility relation can be satisfied depends solely on the existence of read a's previous and immediately adjacent write operation x, and whether that x can satisfy the write-read or write-write pairwise constraints of visibility.

The write-read case is straightforward: if a write a is visible to a read b, then b must have viewed a and an arbitration relation must exist between a and b to ensure the non-deterministic viewed

relation has a linearization point, i.e., to cut off one of the possible paths in the viewed relation. We next define visibility between write operations. We call this propagation path write arbitration: whether two writes are visible to each other depends on the existence of an arbitration relation between them. Writes not ordered by visibility are considered invisible to each other [48] and our proposed write arbitration resolve conflicts in concurrent and invisible operations.

With the complete visibility definition we encode the aforementioned semantics into SemantiX which are then used to reason about the compatibility between applications and semantics/systems.

4 SemantiX Design

We design the distributed system module (DM) based on the definitions in previous section. There are 2 main ways to interact with our framework, (1) compatibility check and (2) compositional check. Compatibility check is specific to DMs and is meant to assert the pairwise relationship between provided arguments. Compositional checks are specific to applications involving multiple components including DMs and are built upon the compatibility check. A DM represents a single distributed system with specific semantics and SemantiX can perform individual DM validation to check if all semantic constraints are sound and compatible, as well as pairwise checks between two DMs to determine if one is satisfiable under another's constraints. More importantly, SemantiX can model applications that rely on multiple distributed systems with different consistency semantics (i.e., DMs) and can check for constraint satisfiability between application logic and assigned semantics by performing a compositional check using the AppGraph.

4.1 Compatibility Check

SemantiX is designed to reason about compatibility among consistency semantics. We define (pairwise) compatibility between two consistency semantics using DMs M_1 and M_2 using an implication-based criterion. M_1 is considered compatible with M_2 if the formula $M_1 \implies M_2$ is valid, i.e., there is no execution that satisfies all constraints of M_1 without also satisfying M_2. We implement this by asserting the negation $\neg(M_1 \implies M_2)$ and checking if it is unsatisfiable using our SMT-based approach. If no counterexample can be found, it implies that M_1 refines/subsumes M_2. This compatibility check is not symmetric: $M_1 \implies M_2$ holding does not necessarily mean that $M_2 \implies M_1$ also holds.

Table 1 shows a result of pairwise compatibility checks for base semantics in SemantiX which in part shows that our consistency models' compatibility faithfully matches with the theoretical results in Figure 1 and in previous literature [43].

To combine multiple consistency models into a single stronger model, we used logical conjunctions on each DM's constraints. By taking the constraints representing each model's semantics and forming their conjunction, we derive a aggregated model that enforces all included constraints simultaneously. This approach allows incremental composition: users can start from a base model and iteratively strengthen it by adding new sets of constraints, either from our implemented models or user-defined constraints, representing additional consistency guarantees.

4.2 AppGraph and Compositional Check

Applications that rely on multiple distributed systems with different consistency semantics can be checked for correctness by performing a compositional check using the AppGraph. Our compositional check verifies if a distributed application, modeled as a graph (*i.e.*, AppGraph) can have its components interact while respecting their individual consistency requirements and guarantees.

The AppGraph (e.g., Figures 5, 6 and 7) consists of *nodes* with two fields: *needs*, required constraint tuples from dependent components, and *provs*, constraint tuples provided to other nodes. Each

node's semantics can be expressed using (1) the predefined consistency DMs and/or (2) user-defined/application-specific constraints that implement the same interface. Edges connect nodes and carry *cons*, constraint tuples for that node pair's interaction, where each tuple is a conjunction of constraints.

In the implementation (also see Fig. 4), we perform a depth-first search (DFS) starting from a given source node, keeping track of visited edges and the accumulated constraints in `path_premise`, along the current path. Constraints in `path_premise` are invisible to users and are dynamic in nature based on which edge(s), `needs`, `provs` are selected. Constraints are populated in and out based on the satisfiability of the previous levels, and current level checks. SemantiX perform compatibility checks, `compatible(check_provs, compose(check_needs, check_ec), path_premise)`: for each edge (u, v) from the current node u, it checks if `provs` from v and `cons` from edge (u, v) satisfy needs from u. This checks if the chosen path satisfies the guarantees required by the destination (`check_provs`) is implied by the conjunction (the result returned from call to `compose`) of the chosen needs of the source (`check_needs`) and the chosen edge constraints (`check_ec`), considering the constraints already accumulated (`path_premise`). If all checks succeed, the application is deemed compatible with the plugged-in DMs and the application logics are also sound as a whole.

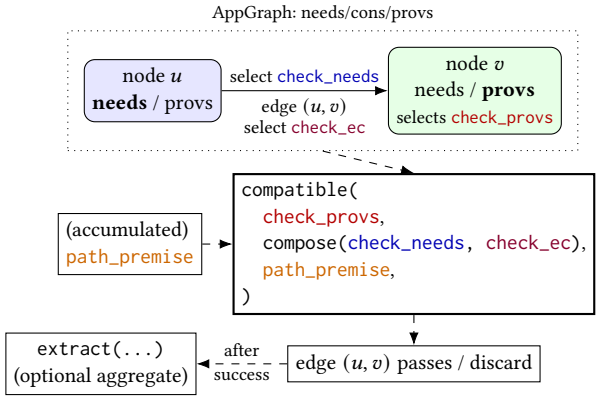

Fig. 4. Example flow for a single edge (u, v): inside AppGraph, select `check_needs`, `check_ec`, and the node's `check_provs`; outside, evaluate `compatible` on `compose(check_needs, check_ec)` and `path_premise`. After a successful compatibility check, `extract` function can optionally aggregate `compatible` return value. For multigraphs, repeat per parallel edge (or edge combinations) with its own cons tuple. Solid arrows are part of AppGraph and have the same meaning as the figures below. Dashed arrows represent control flow.

To avoid redundant checks in case of a subset of nodes and edges needs to be reused for scalability, SemantiX can aggretate a subgraph into a single logical node using the `extract` function. After a successful compositional check, `extract` processes the resulting graph representing a valid assignment of constraints, and outputs a single summary node. This node encapsulates the net semantic guarantee of the subgraph reachable from the `entry` node, provided there are no external edges crossing the subgraph boundary. The function performs a DFS starting from `entry` to compute this summary. For each edge (`src`, `dst`) traversed in the DFS, the extraction function uses the specific `provs` constraint assigned to `src` in the result graph, uses the specific needs constraint assigned to `dst` in the provided graph, and retrieves `cons` constraint assigned to the edge in the argument graph. The extract function creates a composite constraint for this single edge interaction: `compose(src_provs, edge_cons, dst_needs)`. This represents the conjunction of constraints relevant to this specific step in the flow. It collects these composite constraints from all edges reachable from subgraph root in the DFS. Finally, it returns the logical AND of all collected composite constraints. Simply put, axioms and relations define constraints on operations; similarly, `extract` defines a summary constraint on a successfully composed subgraph. If the compositional check found a valid way for components A, B, and C to interact starting from A, `extract(A, graph)`—the first argument is aggregated subgraph's root node and the second is the

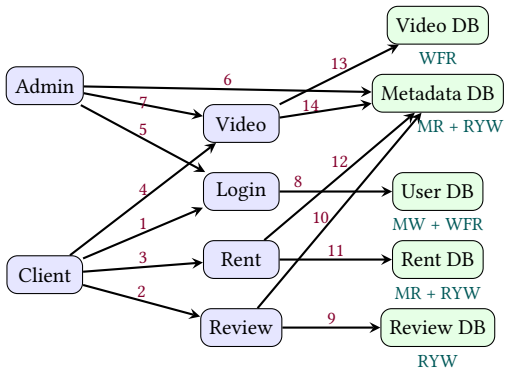

Fig. 5. Architecture of the media streaming service with consistency guarantees. The numbers denote application constraints/flow, node labels on storage systems show provided consistency. Edges: 1–4: Client requests to services (login, review, rent, video). 5: Admin login. 6: Admin updates metadata. 7: Admin uploads video. 8: Login service accesses User DB. 9: Review service writes to Review DB. 10: Review checks content existence in Metadata DB. 11: Rent updates Rent DB. 12: Rent checks content availability in Metadata DB. 13: Video service accesses Video DB. 14: Video service verifies metadata before playback.

return result from `compose` call, containing the full graph's nodes and edges—generates a single formula representing the overall guarantee provided by the A $\longrightarrow$ B $\longrightarrow$ C chain, as seen from A's perspective, based on the specific choices made during the compositional check. By replacing a verified subgraph with its summary node, SemantiX enables modular verification. Subsequent checks on the full application graph can treat this summary as a primitive component, thereby bypassing the internal complexity of the original subgraph and improving solver scalability.

4.3 Compatible Combination Search

SemantiX is capable of searching for compatible DMs for an application given an AppGraph. When blank DMs are plugged into the AppGraph, SemantiX searches for compatible combinations of DMs for the user. Considering that strictly weaker distributed semantics can be replaced with strictly stronger ones and the weaker semantics are typically more performant and cheaper to maintain, SemantiX searches the space in an order that prioritizes weaker semantics. Specifically, we give scores 0 to 5 to semantics in the columns from left to right in Figure 1 and test the combination that has lower sum of scores first during the search. If compatible combinations are found, the search can stop early; replacing a DM in a compatible combination with a strictly stronger one is always guaranteed to be compatible. However, our search strategy is not optimal and investigating more efficient search algorithms remains as future work.

5 Case Study

To demonstrate SemantiX's utility, and to detail how a complex distributed application can be specified in SemantiX for the compatibility check with pluggable distributed services, we present case studies of three applications. Two applications, a movie streaming service and an online shopping mall, are from a microservice benchmark suite [17] and the other is from an academic system that designs causally consistent storage service across multiple weakly consistent distributed storage services [4]. We mainly focus on the movie example to delve into the details. Our framework is designed for a quick specification of applications and prompt compatibility checks, and assumes users to be familiar with axiomatic semantics and SMT encodings which are becoming more common and widely taught in schools [10, 13, 28, 42].

5.1 Movie Streaming Service

Our movie streaming service application [17] involves multiple functionalities that can be represented as regular nodes and edges in AppGraph and distributed storage services (or DBs) that can be captured as DMs (Figure 5).

System Architecture The media streaming service consists of eleven interconnected components, each with specific consistency requirements based on their functional roles. While users can choose a DM for the DB components, they must specify the constraints for the other components.

(1) Client and Admin: These components represent the entry points for regular users and administrative personnel, respectively. They don't enforce specific consistency guarantees themselves but rely on the guarantees provided by the services they interact with.

(2) Authentication: Involves two nodes for clients and admins to authenticate.
- **Login Service**: Manages user authentication and session handling. It requires order preserving writes for correct credential update and the user's updates (e.g., subscription status change) must always be visible for flawless service.
- **User DB**: Stores user credentials and profile information.

(3) Content Management: Movie-related contents made accessible to clients by the admin.
- **Video Service**: Handles video streaming requests and requires users to observe their own prior operations to ensure consistent content delivery.
- **Metadata DB**: Stores information about available videos (titles, descriptions, genres, etc.).
- **Video DB**: Stores the actual video contents.

(4) User-Centered Functions: Mostly hosts client-driven activities.
- **Review Service**: Manages user reviews; reviews should be immediately visible to the writer.
- **Review DB**: Stores user reviews.
- **Rent Service**: Handles media rental transactions; users must see their activities immediately.
- **Rent DB**: Records rental transactions.

Constraint Encoding. We focus on the login and review services in Figure 5 to explain how constrains are encoded in SemantiX. In this example, the application requires that all users must register for an account before they can log in, this would result in a request from client to login service (edge 1), then to user database (edge 8). If users want to post a review (edge 2), they must be logged in (edge 1), and the review service must verify that the title exists in the metadata database (edge 10) before allowing the review to be written (edge 9). This represents a clear dependency chain in the media service example. This dependency chain creates a cascading effect throughout the system where each operation builds upon the successful completion of its prerequisites. The registration-before-login dependency establishes user identity, the login-before-content-access dependency establishes authorization, and the content-validation-before-review dependency establishes referential integrity.

We next show the interaction flow from user login through posting a review and formalize the corresponding edge constraints. Semantically, 6 nodes are interacting with each other (client, login service, user database, review service, metadata database, and review database) in the aforementioned dependency chain. The edges between these nodes represent the flow of API requests and we encode the prerequisites of each request as constraints on edges. Note that all constraints on edges can be directly added to the nodes. However, when modeling services with complex interactions with cross storage system dependencies, edge constraints can clearly separate the providing/requiring constraints from downstream service logic. In the implementation, we defined multiple operations with read or write type to represent interactions between client, service nodes, and storage nodes. Formally, we can represent client's login operation (edge 8) as follows:

$$EC_{\text{login}} \triangleq \forall op_{\text{login}} \in H_{\text{rd}} : \exists op_{\text{reg}} \in H_{\text{wr}} : (op_{\text{login}}, op_{\text{reg}}) \in ob \quad \wedge \quad (op_{\text{reg}}, op_{\text{login}}) \in vis.$$

Since all interactions to User DB pass through Login, and Login node do not have cross edges to outside graph components other than User DB, the subgraph Login, User DB and edge 8 can be aggregated with `extract` function.

For services initiated from the client side, we modeled the behaviors as a set of reuseable common constraints with the return-before binary relationship. Formally:

$$EC_{\text{auth}} \triangleq (op_{\text{login}}, op_{\text{rdreview}}) \in rb \quad \wedge \quad (op_{\text{login}}, op_{\text{wrreview}}) \in rb$$
$$\wedge \quad (op_{\text{login}}, op_{\text{rent}}) \in rb \quad \wedge \quad (op_{\text{login}}, op_{\text{watch}}) \in rb.$$

The common constraints are then added to each edge that is dependent on the login service or refer to the login operation, ensuring the chain of dependencies is respected throughout the system.

For posting reviews, the review service must first verify that the content exists in the metadata database before allowing the review to be written. Similar to the login operation, we can represent the review service's interaction as a set of constraints with existential quantification, object constraints, and return-before relationships, on top of consistency semantics encoded at each nodes' provide and need fields. Formally,

$$EC_{\text{review}} \triangleq \forall op_{\text{review}} \in H_{\text{wr}} : \exists op_{\text{rdmeta}} \in H_{\text{rd}} :$$
$$(op_{\text{review}}, op_{\text{rdmeta}}) \in ob \quad \wedge \quad (op_{\text{rdmeta}}, op_{\text{review}}) \in rb.$$

Following these practices one can specify relatively complex and construct AppGraph with DMs and application-level constraints. AppGraph construction is straightforward, we instantiate nodes with their `needs`/`provs` and connect them with edges carrying the above constraints. Moreover, depending on system designers' requirements, Login node, User DB, and edge 8 can be collapsed into a single aggregated node by applying `extract` (introduced in Section 2) to the subgraph accumulates both the semantic constraints (MW+WFR) and the application constraint EC_{login} into one summary `provs`/`needs` tuple. This lets us reuse the combined guarantee downstream or use it as a portable node in another AppGraph.

Requirements and Compatibility Verification. Each component in the system requires consistency guarantees from the DBs:

- The User DB provides MW and WFR guarantees, ensuring that credential updates are properly sequenced and visible, which is critical for authentication security.
- The Metadata DB provides MR and RYW guarantees, balancing immediate visibility of administrative updates with tolerance for some staleness in user reads.
- The Review DB provides RYW guarantees, prioritizing the user experience of seeing one's own reviews immediately while accepting eventual consistency for other users' views.
- The Rent DB provides MR and RYW guarantees, ensuring consistent transaction records for both administrative and user views.
- The Video DB provides WFR guarantees, maintaining version consistency for content delivery.

The edges in our model represent not just data flow but also critical operational constraints. For example, the edge from the Client to the Login Service carries the constraint that "User must register before login", enforced through a visibility relationship between registration and login operations. Similarly, the edge from the Review Service to the Metadata Database enforces the constraint that "Check metadata before reading reviews", ensuring that reviews are only accessible for valid content.

When we apply the compositional check to this system, starting from the client node, it successfully verifies that the composition of these services with their respective consistency guarantees satisfies the overall system requirements. This verification confirms that despite the heterogeneous consistency guarantees across different components, the system as a whole maintains its required properties. For instance, the login flow correctly enforces that registration must precede login, and the review system correctly ensures that users can immediately see their own reviews.

5.2 Online Shopping Mall

An e-commerce application [17] is similar to the movie streaming service but its architecture and constraints are simpler (Figure 6).

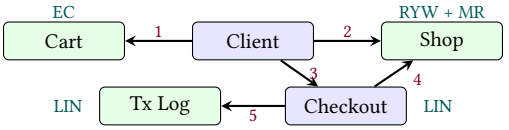

Fig. 6. Architecture of the online shopping system with consistency guarantees. Numbers denote application constraints/flow on edges: 1: client read/write cart; 2: client browse shop; 3: client purchase via checkout; 4: checkout check/update shop; 5: checkout record to Tx Log. Node labels show provided consistency semantics.

Architecture. The online shopping application consists of five key components and only models client behaviors:

(1) Client: The end-user interface through which customers browse products, manage their shopping cart, and initiate purchases. The client itself doesn't enforce specific consistency guarantees but relies on guarantees provided by the interacting services.

(2) Cart DB: The shopping cart DB maintains customers' shopping cart state. It stores items that customers intend to purchase and supports read and write operations. The cart can be loosely managed and immediate propagation of updates is not required.

(3) Shop DB: Maintains the authoritative inventory state for all products. Clients can observe stale state, but the shop must provide up-to-date information to the checkout service which process purchase transactions.

(4) Checkout: Acts as a serialization point for purchase transactions, providing linearizability guarantees. This critical component ensures that concurrent purchase requests are processed in a well-defined sequence, preventing race conditions that could lead to inventory inconsistencies or duplicate transactions. Even if it works with DBs that guarantee weaker semantics than linearizability, its logic should guarantee strong consistency.

(5) Tx Log DB: Records all completed purchases with linearizability guarantees, ensuring that transaction records are stored in a strictly consistent manner for auditing and financial reconciliation purposes.

Requirements and Compatibility Verification The consistency requirements of this system vary by component:

- The shop DB provides RYW and MR, which are sufficient for a centralized Checkout service to manage inventory. RYW ensures that once the Checkout updates inventory levels, any subsequent reads will observe these updates. MR ensures that if a client observes a certain inventory state, all subsequent reads will observe either that same state or a more recent one, but seeing outdated state is okay as Checkout will prevent ill-formed purchase transaction (e.g., purchasing out-of-stock item).
- Checkout and transaction log DB both provide linearizability, the strongest consistency model, ensuring that all operations appear to execute atomically in a sequential order that is consistent with the real-time ordering of those operations. This is critical for the financial transactions.
- The cart service has more relaxed consistency requirements, as temporary inconsistencies in cart state don't impact the integrity of the inventory or financial records.

An interesting aspect of this example is that Checkout can guarantee linearizability as long as it is the sole entry updating the Shop DB. The verification confirms that Checkout's linearizability guarantee is sufficient to coordinate between the shop's RYW+MR guarantees and the transaction log's linearizability, ensuring that inventory updates and transaction records remain consistent even under concurrent operations.

5.3 Cross Causal (XC) Storage

Cross causal (XC) storage [4] combines multiple distributed storage services, each offering only a basic consistency guarantee, to provide causal consistency as a whole (Figure 7). As XC storage itself is a system, its constraints are very different from application-level constraints in previous case studies; XC storage showcases the capability of SemantiX beyond modeling application logics.

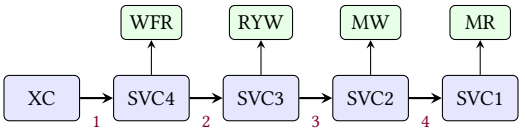

Fig. 7. Cross-service causal composition chain. Edge labels mark the application-level semantic "glue" added per hop, e.g., XC→SVC4 supplies the missing guarantees (MR+MW+RYW) to reach causal consistency, and the downstream edges do likewise for their missing session guarantees. Per-level storage nodes provide guarantees for each service layer.

Users accessing data through the XC node get causal consistency, but individual storage DB maintains weaker semantics than causal. SVC4 node's storage backend only provides WFR. To achieve causal consistency, application places a shim layer to strengthen the semantics in the design and this is imposed on the edge between SVC4 and XC. While the SVC4-XC combination can achieve causal consistency, SVC4's downstream service SVC3's storage backend only provides RYW. To achieve cross-system causal consistency, the augmentation continues down the chain.

The goal here is to verify whether their composition, mediated by specific inter-service constraints, symbolic isolation, and the role of explicit "glue" constraints.

Architecture. The architecture forms a chain XC → SVC4 → SVC3 → SVC2 → SVC1. Each service SVCn ($n = 1..4$) is designed to provide only *one* of the four session guarantees via its `provs` attribute: SVC1 provides MR, SVC2 provides MW, SVC3 provides RYW, and SVC4 provides WFR.

Constraints Encoding and Compatibility Verification. To prevent logical interference, the guarantee provided by each service and the constraints added on the edges are instantiated using unique symbols specific to that service or edge. This ensures, for example, that the MR constraints of SVC1 only apply to operations symbolically designated as belonging to SVC1.

The composition logic resides in the "glue" constraints defined on the `cons` attribute of the edges connecting the services. Since each service node only provides one session guarantee, the edge SVCn → SVC$(n+1)$ must provide constraints representing the other three session guarantees needed to potentially satisfy causal consistency up to that point.

The core idea of this example is to build a stronger guarantee by composing weaker guarantees provided by individual services (SVC1 . . . 4), using the edges to supply the missing pieces which reflects the application design as shown in Figure 7.

For this composition to be meaningful in the SMT-solver-based SemantiX, we must conceptually link operations across services that refer to the same logical action or data. For instance, if write w_1 in SVC1 and write w_2 in SVC2 both represent updates to the same user profile, "glue" constraints equating relevant aspects (like w_1.obj = w_2.obj or w_1.ival = w_2.ival despite their different symbolic names) is necessary. While the current test code simplifies this by focusing on composing the consistency axioms, a more rigorous model would include these explicit symbol equivalence constraints on the edges (we are asserting symbols of w_1 are identical to those of w_2 for simplicity).

After a single logical formula representing the net end-to-end guarantee achieved by the SVC chain is synthesized starting from SVC1 to XC, the formula is then compared against a formal definition of CC to verify compatibility, which succeeds. While the original XC implementation [4] ties together only EC storage services, this example further explores more diverse storage services

and demonstrates the flexibility of SemantiX. With the successful compositional check, the multi-node XC chain can be collapsed into a single entity (similar to how Login + User DB could be summarized) by applying `extract` to the full XC graph. This yields one aggregated `provs/needs` tuple representing the composed cross storage system causal consistency guarantee for any future downstream use.

6 Evaluation

We discuss our experience in using SemantiX and its performance in checking compatibilities.

Experience We implemented SemantiX in Python on Z3 version 4.15.1. It took approximately 3.7K lines of code to implement SemantiX. To use SemantiX, users supply AppGraph whose nodes and/or edges carry built-in or custom DMs that extend the base model and relation interfaces. When multiple DMs are attached, SemantiX checks for compatibility of given input or exhaustively explores valid combinations.

	EC	MR	MW	RYW	WFR	PRAM	CC	LIN
EC	-	24	22	26	25	24	35	32
MR	23	-	12	16	75	16	26	217
MW	27	16	-	18	16	16	23	22
RYW	193	106	45	-	59	20	29	26
WFR	24	125	12	17	-	18	44	54
PRAM	24	14	10	14	178	-	27	28
CC	33	20	16	19	19	20	-	282
LIN	26	21	17	22	22	22	44	-

Table 1. Pairwise check performance. Semantics in the first column are checked if they satisfy semantics in other columns. Each cell shows the execution time for completing the check in milliseconds. The blue text indicates the check result is true and red means false.

Regarding effort and the learning curve, once semantic requirements are understood, encoding an AppGraph is straightforward. Our case studies only require 183, 80, and 252 lines of code for encoding the movie streaming service, online shopping mall, and XC storage, respectively, thanks to existing facilities such as the DMs in the SemantiX framework. These relatively small efforts for constraint encoding and automated checks validates our framework's goal of enabling agile compatibility checks compared to typical end-to-end formal verification which could have huge proof-to-code/specification ratio [19, 20, 23, 37, 39] which could follow with confidence after our SemantiX check passes.

The scalability of our tool is mainly expressed via modular composition, AppGraph nodes and edges compose modularly: large systems can be decomposed into sub-AppGraphs, verified independently, and re-aggregated into monolithic node that preserves the verified properties of its components. This pattern (*e.g.*, Case Study 2 in Section 5) also enables parallel SMT solving on sub-components and reuse of solved results for large service meshes.

In general, our framework offers early compatibility detection, architecture validation before implementation, and pre-verification screening to de-risk downstream formal proofs. It targets teams building correctness-critical distributed systems who want a high-level, low-maintenance checker rather than line-by-line deductive proof development (*e.g.*, using tools like Verus[27], Dafny[29], Rocq[46], and Lean[36]).

Performance SMT solvers often take an extended period to complete their tasks. Thus, we measure the duration of the compatibility check in SemantiX. All experiments were conducted on an Intel i7-1360P processor with 64GB RAM running NixOS 25.11 (Linux 6.12.30).

Pairwise Compatibility Check. First, we evaluated the compatibility between pairs of base consistency semantics, EC, MR, RYW, MW, WFR, PRAM, CC, and LIN, in SemantiX. Table 1 shows the execution times for these pairwise compatibility checks and the results of the check encoded in colors (blue: compatible, red: non-compatible). On average, it took only 31 ms per check and the results of the checks are correct with respect to the theory.

Compatible DM Search. Next, we evaluate SemantiX using three case studies in Section 5. This evaluation focuses on two key metrics: the framework's ability to find valid compositions with the weakest semantics (i.e., the minimum score combination in Section 4.3) and its performance characteristics under different system complexities. For each case study, we gave blank DMs to each application so that SemantiX search and output compatible DM combinations. SemantiX uses the search strategy in Section 4.3. The results are summarized in Table 2.

Application	# of combs.	All	Min score	Single
Movie	17^5 = 1.4M	64.82 hr*	20.43 sec	16.5 ms
Shopping	17^3 = 4.9K	251.77 sec	1.65 sec	5.12 ms
XC store	17^4 = 83.5K	3.98 hr	261.09 sec	17.17 ms

Table 2. Search time for compatible DMs. Columns show number of searched DM combinations, time to search all combinations (all), time to find the minimum score combination (min score), and average time to check one combination (single). (* predicted time)

Movie streaming service requires finding DMs corresponding to five DB nodes that satisfy application node and edge constraints. Given that SemantiX contains 17 DMs as illustrated in Figure 1 the full search space contains 17^5 combinations. On average, checking each combination takes 16.5 ms and can find a compatible combination with the lowest score in 20.43 seconds with a timeout of 10 seconds for each check. Checking the entire search space is feasible but is expected to take 64.82 hours (we have completed 800,507 checks or 56.28% of search space in 36.55 hours).

The online shopping mall example is simpler than the movie streaming service and involves only three DB nodes to search. We evaluated all 17^3 possible combinations of consistency semantics across these nodes. The evaluation completed in 251.77 seconds with an average duration of 5.12 ms per combination with 0 timeouts. The minimum score combination was found in 1.65 seconds.

Finally, four DB node combinations (i.e., 17^4 cases) are searched for the cross causal storage. The total search completes in 3.98 hours with an average duration of 17.17 ms per combination. Our tool successfully identified the combination in our original design (also the minimun score combination) in about 261.09 seconds. The timeout rate is only about 0.15% of cases.

Overall, depending on the complexity of the constraints the compatibility check times vary, but individual checks are done in less than 20ms on average. Even though searching through compatible answers in the entire search space could take extended time, finding the optimal, minimum-score configuration for each case study takes only less than 5 minutes.

7 Related Work

Our work sits at the intersection of distributed systems, formal verification, and automated reasoning. We organize the related work into consistency verification frameworks, SMT-based approaches for distributed systems, and compositional analysis tools.

Several frameworks have been developed to verify consistency properties in distributed systems. Quelea [40] provides a declarative programming model where developers specify consistency requirements as contracts using a shallow extension of Haskell and discharge proof obligations to SMT solvers. The system then automatically selects appropriate consistency levels. However, Quelea focuses on single-system consistency rather than cross-service composition and relies on conservative approximations of happens-before relationships due to limitations in expressing transitive closures in first-order logic.

Jepsen [22] takes a different approach by performing black-box testing of distributed systems to detect consistency violations through controlled fault injection. While effective at finding bugs in real systems, Jepsen cannot provide formal guarantees about system compositions or verify that specific consistency requirements are met by design.

The use of SMT solvers for distributed systems verification has gained traction in recent years. Kuper et al. [24] argue for domain-specific solvers embedded in solver-aided languages like Liquid-Haskell and Rosette to build consistent-by-construction applications. They identify key challenges including CRDT verification, message reordering analysis, and transitive closure reasoning. While they advocate for specialized solvers, our work demonstrates that careful encoding strategies can make general-purpose SMT solvers effective for consistency verification.

Fan et al. [14] develop an ordering consistency theory for verifying multi-threaded programs under memory models like sequential consistency and TSO. Their approach uses incremental consistency checking and specialized theory propagation to efficiently handle ordering constraints. Although focused on memory models rather than distributed consistency, their formalization of ordering relations ($\prec_{rf}$, $\prec_{fr}$, $\prec_{ws}$) provides insights for our visibility-based encoding of session guarantees.

MixT [35] is a domain specific language to simplify programming mixed-consistency transactions over multiple distributed systems. It uses information flow theory to generate transactions consisting of sub-transactions that safely execute across multiple systems with different semantics. While MixT is a specialized tool to generate safe and efficient transactions for given consistency semantics of distributed systems, SemantiX allows developers to check correctness of their operations over chosen combination of distributed systems or search for compatible consistency semantic combinations.

SemantiX differs from existing approaches in several ways: unlike single-system verification frameworks, we focus specifically on the composition problem where services with different consistency models work together; unlike testing-based approaches, we provide formal compatibility checks over different semantics; unlike theorem-proving approaches, we use automated reasoning to make the verification process more accessible to practitioners.

Our encoding strategy addresses limitations identified in prior work by avoiding explicit reasoning about global event orderings and instead treating consistency as logical constraints over abstract executions. This approach makes the verification tractable while remaining expressive enough to handle realistic service compositions. Our framework's ability to synthesize end-to-end guarantees from component specifications provides developers and system architects with actionable insights about their systems' consistency behavior.

8 Conclusion

In this paper, we presented SemantiX, the first automated framework for checking semantic compatibility between applications and compositions of distributed systems with different consistency guarantees. Our key contributions include formal definitions of distributed consistency semantics including the first rigorous formal definition of visibility, the AppGraph modeling approach for systematic encoding of microservice-like applications, and an SMT-based compatibility checking framework that can check compatibility between given distributed semantics and applications and search for compatible combinations of distributed semantics for the application. Through case studies on three distributed applications, we demonstrated that SemantiX can model and verify complex service compositions with little encoding effort while providing formal guarantees about system correctness. As distributed system architectures grow increasingly complex, SemantiX provides developers with essential tools for quickly checking the integrity of the distributed application designs enabling more agile system design where components can be modified or replaced with confidence.

Data Availability

The data and code that support the findings of this study are currently in the supplementary materials section in the HotCRP submission site accessible to the reviewers. Upon acceptance of the paper, we will make the materials openly accessible in a long term archival digital repository.

Acknowledgments

We would like to thank our anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grant 2442888.

References

[1] 2024. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db/.

[2] 2024. Sequential consistency without borders: how D1 implements global read replication. https://blog.cloudflare.com/d1-read-replication-beta/.

[3] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distrib. Comput.* 9, 1 (March 1995), 37–49. doi:10.1007/BF01784241

[4] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 761–772. doi:10.1145/2463676.2465279

[5] Roberto Baldoni, C. Sparziani, Sara Tucci Piergiovanni, and Daniela Tulone. 2002. An Implementation of Causal Memories using the Writing Semantic. In *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002 (Studia Informatica Universalis, Vol. 3)*, Alain Bui and Hacène Fouchal (Eds.). Suger, Saint-Denis, rue Catulienne, France, 41–50.

[6] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I.* Springer-Verlag, Munich, Germany, 415. doi:10.1007/978-3-030-99524-9_24

[7] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. 2004. From session causality to causal consistency. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings.* 152–158. doi:10.1109/EMPDP.2004.1271440

[8] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. 2004. Session Guarantees to Achieve PRAM Consistency of Replicated Shared Objects. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–8.

[9] Sebastian Burckhardt. 2014. *Principles of Eventual Consistency* (principles of eventual consistency ed.). Foundations and Trends® in Programming Languages, Vol. 1. Now Publishers. 1–150 pages. https://www.microsoft.com/en-us/research/publication/principles-of-eventual-consistency/

[10] Antonio Cerone, Markus Roggenbach, James Davenport, Casey Denner, Marie Farrell, Magne Haveraaen, Faron Moller, Philipp Körner, Sebastian Krings, Peter Csaba Ölveczky, Bernd-Holger Schlingloff, Nikolay Shilov, and Rustam Zhumagambetov. 2021. Rooting Formal Methods Within Higher Education Curricula for Computer Science and Software Engineering — A White Paper —. In *Formal Methods – Fun for Everybody*, Antonio Cerone and Markus Roggenbach (Eds.). Springer International Publishing, Cham, 1–26.

[11] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1, 2 (aug 2008), 1277–1288. doi:10.14778/1454159.1454167

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Budapest, Hungary, 337.

[13] Brijesh Dongol, Catherine Dubois, Stefan Hallerstede, Eric Hehner, Carroll Morgan, Peter Müller, Leila Ribeiro, Alexandra Silva, Graeme Smith, and Erik de Vink. 2024. On Formal Methods Thinking in Computer Science Education. *Form. Asp. Comput.* 37, 1, Article 8 (Dec. 2024), 23 pages. doi:10.1145/3670419

[14] Hongyu Fan, Zhihang Sun, and Fei He. 2023. Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models. *ACM Trans. Program. Lang. Syst.* 45, 1 (March 2023). doi:10.1145/3579835

[15] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. 2023. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *Proceedings of the 29th Symposium on Operating Systems*

Principles (SOSP '23). Association for Computing Machinery, Koblenz, Germany, 298–313. doi:10.1145/3600006.3613176

[16] Bernd Finkbeiner and Sven Schewe. 2007. SMT-based synthesis of distributed systems. In *Proceedings of the Second Workshop on Automated Formal Methods* (Atlanta, Georgia) *(AFM '07)*. Association for Computing Machinery, New York, NY, USA, 69–76. doi:10.1145/1345169.1345178

[17] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, Providence, RI, USA, 3–18. doi:10.1145/3297858.3304013

[18] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic. *Proc. ACM Program. Lang.* 5, POPL, Article 42 (jan 2021), 29 pages. doi:10.1145/3434323

[19] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 653–669.

[20] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428

[21] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (7 1990), 463–492. doi:10.1145/78969.78972

[22] Kyle Kingsbury, Kit Patella, Raphael 'kena' Poss, Brandon Mitchell, Jeff Smick, Daniel Mai, UncP, Nurture Nature, Stevan A, Timur Yusupov, Ben Darnell, Alex Abdugafarov, Neil Shen, Jacques Fuentes, Ellen Marie Dash, Manish R Jain, Jacob, Cwen Yin, Vojtěch Juránek, Andrei Dan, Mikhail Filatov, Pavel Lipskii, Akihiro Suda, Alexander Abramov, Pavlo Baron, Martin Martinez Rivera, siddontang, fruitcupWarrior, Nate Larsen, and Jonathan Halterman. 2025. jepsen-io/jepsen. https://github.com/jepsen-io/jepsen. https://github.com/jepsen-io/jepsen

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. doi:10.1145/1629575.1629596

[24] Lindsey Kuper and Peter Alvaro. 2019. Toward Domain-Specific Solvers for Distributed Consistency. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–14. doi:10.4230/LIPIcs.SNAPL.2019.10 Keywords: distributed consistency, SMT solving, theory solvers.

[25] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. doi:10.1145/359545.359563

[26] Butler Lampson. 2021. Hints and Principles for Computer System Design. arXiv:2011.02455 [cs.DC]

[27] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (apr 2023), 30 pages. doi:10.1145/3586037

[28] Konstantin Laufer, Gunda Mertin, and George K. Thiruvathukal. 2024. Engaging More Students in Formal Methods Education: A Practical Approach Using Temporal Logic of Actions . *Computer* 57, 12 (Dec. 2024), 118–123. doi:10.1109/MC.2024.3462188

[29] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

[30] Richard J. Lipton and Jonathan Sandberg. 1988. *PRAM: A Scalable Shared Memory*. Technical Report TR-180-88. Princeton University.

[31] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. doi:10.1145/2043556.2043593

[32] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation*

(Carlsbad, California) *(OSDI'22)*. USENIX Association, Berkeley, CA, USA.

[33] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. 2025. Deriving Semantic Checkers from Tests to Detect Silent Failures in Production Distributed Systems. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*. USENIX Association, Boston, MA, USA.

[34] Kai Ma, Cheng Li, Enzuo Zhu, Ruichuan Chen, Feng Yan, and Kang Chen. 2024. Noctua: Towards Automated and Practical Fine-grained Consistency Analysis. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, Athens, Greece, 704–719. doi:10.1145/3627703.3629570

[35] Mae Milano and Andrew C. Myers. 2018. MixT: a language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 226–241. doi:10.1145/3192366.3192375

[36] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[37] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. doi:10.1145/3158116

[38] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, Jakub Szefer, and Hakim Weatherspoon. 2016. Towards Weakly Consistent Local Storage Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 294–306. doi:10.1145/2987550.2987579

[39] Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. 2019. WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. ACM, New York, NY, USA, 299–311. doi:10.1145/3357223.3362739

[40] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, Portland, OR, USA, 413–424. doi:10.1145/2737924.2737981

[41] Vidhya Tekken Valapil, Sorrachai Yingchareonthawornchai, Sandeep Kulkarni, Eric Torng, and Murat Demirbas. 2017. Monitoring Partially Synchronous Distributed Systems Using SMT Solvers. In *Runtime Verification*, Shuvendu Lahiri and Giles Reger (Eds.). Springer International Publishing, Cham, 277–293.

[42] Maurice ter Beek, Manfred Broy, and Brijesh Dongol. 2024. The Role of Formal Methods in Computer Science Education. *ACM Inroads* 15, 4 (Nov. 2024), 58–66. doi:10.1145/3702231

[43] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, Vol. 0. 140–149. doi:10.1109/PDIS.1994.331722

[44] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 309–324. doi:10.1145/2517349.2522731

[45] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 172–182. doi:10.1145/224057.224070

[46] The Coq development team. 1999 – 2024. The Coq proof assistant. http://coq.inria.fr.

[47] Paolo Viotti, Christopher Meiklejohn, and Marko Vukolić. 2016. Towards property-based consistency verification. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '16)*. Association for Computing Machinery, London, United Kingdom. doi:10.1145/2911151.2911162

[48] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1 (6 2016). doi:10.1145/2926965

[49] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (jan 2009), 40–44. doi:10.1145/1435417.1435432

[50] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. doi:10.1109/ICDE.2018.00044

[51] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4 (12 2018). doi:10.1145/3269981