Welder: Compositional Liveness Verification of Cluster Control Planes

Zhizhen Cathy Cai
University of Illinois
Urbana-Champaign, IL, USA
cathyc5@illinois.edu

Nikhil Date
University of Illinois
Urbana-Champaign, IL, USA
ndate2@illinois.edu

Jiawei Tyler Gu
University of Illinois
Urbana-Champaign, IL, USA
jiaweig3@illinois.edu

Cody Rivera
University of Wisconsin
Madison, WI, USA
cjrivera@wisc.edu

Tej Chajed
University of Wisconsin
Madison, WI, USA
chajed@wisc.edu

Oded Padon
Weizmann Institute of Science
Rehovot, Israel
oded.padon@weizmann.ac.il

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

Xudong Sun
University of Toronto
Toronto, ON, Canada
xudong.sun@utoronto.ca

Abstract

We present the first compositional approach to formally verifying cluster control planes such as Kubernetes. Control planes are large, distributed systems composed of interacting controllers with subtle liveness and safety dependencies. Our approach, Welder, emphasizes compositionality to make verification manageable and scalable. Welder makes threefold contributions on specification, proof, and implementation. First, Welder provides a general specification called *COmpositional REconciliation (CORE)*. CORE states that a set of controllers collectively reconcile the cluster state correctly, precluding bugs in both individual controllers and cross-controller interactions. CORE is an open specification that enables compositional verification: if two compatible sets of controllers each implement CORE, so does their composition. Second, Welder introduces new proof techniques for compositional liveness reasoning about controller interactions, including liveness dependencies and interference. Third, we mechanize Welder as a framework and use it to implement and verify a fraction of the Kubernetes control plane—three core controllers and one custom controller. The verified controllers can readily be deployed in Kubernetes platforms with comparable performance to (unverified) official ones. Welder enables a progressive way to verify cluster control planes by gradually verifying that each controller implements CORE.

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843868

CCS Concepts: • **Computer systems organization** → **Reliability**; **Cloud computing**; • **Software and its engineering** → **Software verification**.

Keywords: Kubernetes, formal verification, liveness verification, compositional verification, cloud, reliability

ACM Reference Format:
Zhizhen Cathy Cai, Nikhil Date, Jiawei Tyler Gu, Cody Rivera, Tej Chajed, Oded Padon, Tianyin Xu, and Xudong Sun. 2026. Welder: Compositional Liveness Verification of Cluster Control Planes. In *Symposium on Operating Systems Principles (SOSP '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3830418.3843868

1 Introduction

Cluster control planes such as Kubernetes [33], Borg [81], and Twine [76] are among the most critical system software of modern clouds. They manage large-scale cluster resources and deployed applications. A control plane is composed of a fleet of interacting controllers running as loosely coupled microservices. Modern cluster control planes are highly extensible: users can implement and deploy new controllers that interact with the existing ones. In Kubernetes, for example, core controllers implement general management logic, while a thriving ecosystem of thousands of custom controllers extends Kubernetes with domain-specific resource and application management logic.

Ensuring the correctness of cluster control planes is critical but challenging. Control planes implement *state reconciliation*: they monitor the cluster state and continuously reconcile the *current* cluster state to match a *desired* state. Controllers implement domain-specific state reconciliation logic for managing different resources and applications and collectively manage the cluster state. Correctness is hard because there are subtle liveness and safety dependencies among

controllers. Prior work shows that interference among controllers causes oscillation failures where the cluster state never enters a stable desired state—a serious liveness violation [57, 75]. The interference is harder to avoid when the deployed controllers are developed independently by different teams [77]. In addition, controllers need to tolerate failures and asynchrony that are common in distributed systems. Bugs in cluster control planes have caused catastrophic failures [3, 10, 11, 14, 61].

We present Welder, the first compositional approach to formally verifying cluster control planes, with three major contributions. First, we present *COmpositional REconciliation (CORE)*, a general, compositional specification for cluster control planes. CORE includes both liveness and safety components: liveness captures correct state reconciliation logic, and safety restricts inter-controller interference. Second, we design new compositional liveness proof techniques for controller interactions, including liveness dependencies, conflicts, and interference. Third, we implement Welder as a framework for writing practical controller implementations and machine-checked proofs. We used the Welder framework to build and verify a composition of four practical Kubernetes controllers, including both core and custom controllers.

Formally verifying cluster control planes poses two key challenges: specification and proof. First, we need a specification that captures the essential functionality of a control plane—state reconciliation—and is compositional in two ways: (1) verifying that a large, complex control plane meets the specification should decompose into smaller verification tasks for individual controllers, and (2) the specification must be *open* to incorporating newly verified controllers, because control planes are not closed systems—they are routinely extended with custom controllers. Second, proving that controllers implement the specification requires reasoning about a controller's interactions with other controllers, e.g., how its liveness depends on another controller's liveness. Compositional verification of liveness dependencies is a daunting task. Prior work [75] only proves liveness of individual controllers, and the proofs are not compositional.

We present a general formal specification for cluster control planes, termed COmpositional REconciliation (CORE), written in a hybrid of Temporal Logic of Actions (TLA) [50] and rely-guarantee style [46] (§3). CORE for a subset of a cluster control plane S (a set of controllers) states that S reconciles the cluster state to eventually match the desired state stably. CORE reuses Eventually Stable Reconciliation (ESR), a liveness property proposed by prior work [75], to specify each controller's reconciliation. For compositionality, CORE employs *request-centric* rely-guarantee conditions, written as safety invariants, to restrict inter-controller interference (§3.2). The rely assumes that other controllers never send requests within the scope of S, and the guarantee promises that controllers in S only send requests within their scope. CORE also specifies controllers' liveness dependencies.

CORE enables compositional verification: if two compatible controller sets S_1 and S_2 each implement CORE, then so does their composition. Compatibility essentially means disjoint scopes between S_1 and S_2. Verifying that a control plane implements CORE is done by verifying that each controller (a singleton set) implements its CORE. This compositionality makes CORE an *open specification*, similar to how a cluster control plane is an open system: when a new controller c joins a control plane S, we can verify $S \cup \{c\}$ by proving c's CORE and reusing S's CORE proof.

Compositional verification poses three new proof challenges. The first challenge is reasoning about liveness dependencies. CORE uses ESR as a general liveness formalization of correct state reconciliation, but ESR is seemingly not compositional when considering the common pattern of *iterative updates*. ESR states that the state will eventually be reconciled to the desired state of a controller, assuming the desired state of that controller *remains stable*. In the iterative update pattern, one controller iteratively changes the desired state of another controller it depends on and waits for the state to reconcile. This pattern contradicts ESR's stability premise—we need to know that the state will be reconciled for a desired state that we know is not stable. Welder addresses this challenge with a new proof rule that captures a surprising property of ESR in a compositional setting: even though ESR assumes stability of the desired state, the semantics of ESR for compositional reasoning is stronger—it permits the desired state of a controller c to change if the change comes from a dependent controller *as a result* of c finishing its state reconciliation (§4.1). We use this proof rule to verify the rolling update feature collectively implemented by Kubernetes Deployment and ReplicaSet controllers.

The second challenge is reasoning about conflicts caused by concurrent updates to shared objects. It is common for two controllers to update disjoint parts of the same object to collectively update the cluster state. In version-based systems like Kubernetes, one controller would succeed and increment the version, and the other would fail due to a version conflict. If some controller always loses the race then it cannot make progress. Such an unfair trace cannot be precluded by traditional fairness assumptions like TLA's weak fairness or strong fairness [50]. Welder introduces a transactional abstraction that avoids version conflicts, implements it using existing APIs, and justifies the abstraction based on an appropriate fairness assumption (§4.2). This approach relieves developers from reasoning about conflicts and retries.

The third challenge is reasoning about progress that depends on the absence of disruptive interference. This requires applying different safety invariants on effects of different requests during liveness proof. To reduce the proof burden, Welder provides a new proof strategy that uses a unified state-preservation invariant (§4.3). The invariant states that the cluster state managed by the target controller is preserved by

other controllers. Developers first prove this invariant (using the rely condition), then reason about progress without worrying about interference.

We implement the Welder framework on top of Anvil [75] and Verus [52]. To support compositional verification, Welder introduces a parametric environment model that can be instantiated with any set of controllers, enabling verification of a controller's behavior alongside arbitrary other controllers. Welder also supports verifying *core* controllers, which pose new proof challenges compared to custom controllers due to runtime dynamics and new fault models (§6).

We have used Welder to implement a small but challenging fraction of Kubernetes: three core controllers for managing ReplicaSet, Deployment, and StatefulSet, and a custom controller for managing RabbitMQ; the RabbitMQ controller was ported from Anvil to Welder. The four controllers represent common controller interactions: the Deployment controller has a liveness dependency on ReplicaSet, the RabbitMQ controller has a liveness dependency on StatefulSet, and each controller has a safety assumption (rely condition) on others. We verified that the composition of the four controllers implements CORE. We discovered deep bugs related to controller interactions during verification. The four verified controllers achieve competitive performance compared to existing widely used but unverified Kubernetes controllers.

Summary. We make the following main contributions:

- COmpositional REconciliation (CORE), a general, compositional specification for cluster control plane correctness;
- Several compositional liveness proof techniques for reasoning about controller interactions;
- Welder as a mechanized framework for developing and verifying practical cluster control planes;
- Four representative and practical Kubernetes controllers verified using Welder; and
- An evaluation of the end-to-end correctness and performance of the four verified controllers.

The Welder framework and the verified controllers are available at https://github.com/anvil-verifier/anvil.

2 Background

Modern cluster control planes, including Kubernetes [33], Borg [81], Twine [76], and ECS [64], are designed around the principle of *state reconciliation*. Users declaratively specify a desired state of the cluster, and the control plane continuously reconciles the cluster from its current state toward this target. Concretely, at each iteration of the reconciliation, the control plane checks whether the current cluster state matches the desired state; if not, it performs corrective operations to move the cluster toward the desired state (e.g., creating new replicas of application Pods when existing replicas do not match the desired count). This reconciliation process provides a uniform abstraction for managing dynamic, distributed cluster resources and applications.

```
fn d_reconcile(d: Deployment, r: Option<Response>,
    state: DState) -> (DState, Option<Request>) {
  match state.step {
    Init => { // List all ReplicaSets (RS) in namespace
      return (ManageRS, list_rs(namespace));
    }
    ManageRS => {
      let (newrs, oldrs_lst) = filter(parse(r), d);
      if newrs.is_none() {
        return (CreateNewRS, None);
      }
      if newrs.status.replicas == newrs.spec.replicas {
        if newrs.spec.replicas != d.spec.replicas {
          return (ScaleNewRS, None);
        }
        return (ScaleDownOldRS, None);
      }
      return (Pending, None);
    }
    CreateNewRS => { ... }
    ScaleNewRS => {
      let diff = ...
      let target = newrs.spec.replicas + diff;
      return (Pending, Some(update(newrs{
        spec.replicas: target})));
    }
    ScaleDownOldRS => { ... }
    ... // more step branches are omitted
  }}
```

Figure 1. The `reconcile` function of a Kubernetes Deployment controller [20] written in a state-machine style. It starts from the `Init` state and transitions iteratively until the observed ReplicaSets match the desired state. Each step returns the next state in the state machine and an external request (e.g., a call to Kubernetes API). Each transition is atomic with respect to cluster-state changes. Any `update` on ReplicaSet objects like `newrs` will trigger the ReplicaSet controller (see Figure 2).

In practice, reconciliation is not implemented by a monolithic component, but by a *collection of controllers*, each responsible for a subset of resources. Controllers execute independent reconciliation loops that read and update shared cluster state, represented by state objects, through the API server. This design decomposes cluster management into loosely coupled components, enabling extensibility and modularity. However, it also introduces implicit dependencies: controllers do not operate in isolation but interact indirectly and asynchronously through shared state.

Consider the interaction between the Deployment and ReplicaSet controllers [20, 23] in Kubernetes. Figure 1 shows the `reconcile` function of the Deployment controller structured as a state machine (each transition moves the cluster closer to the target state). The Deployment controller manages application replicas by creating/updating ReplicaSet objects. The ReplicaSet controller observes the created/updated ReplicaSet objects and creates/updates Pods accordingly, as shown in Figure 2. In this workflow, reconciliation is distributed across multiple controllers: the Deployment controller specifies the desired state at the ReplicaSet level, while the ReplicaSet controller asynchronously materializes that state at the level of Pods. Figure 3(a) depicts this process.

```
fn rs_reconcile(rs: ReplicaSet, r: Option<Response>,
    state: RSState) -> (RSState, Option<Request>) {
  match state.step {
    Init => { // List all Pods in namespace
      return (ManagePods, list_pods(namespace));
    }
    ManagePods => {
      let matching = filter(parse(r), rs);
      let desired = rs.spec.replicas;
      if matching.len() < desired { return (
        CreatePods(desired - matching.len()), None);
      }
      if matching.len() > desired { return (
        DeletePods(matching.len() - desired), None);
      }
      return (UpdateStatus, None);
    }
    CreatePods(diff) => { ... }
    DeletePods(diff) => { ... }
    UpdateStatus => { ... }
    ... // more step branches are omitted
  }}
```

Figure 2. The `reconcile` function of a Kubernetes ReplicaSet controller [23]. The `update` of the ReplicaSet object `newrs` in Figure 1 will trigger `rs_reconcile` and the ReplicaSet object updated (or created) by the Deployment controller is the first argument `rs` of the `rs_reconcile` function.

The asynchronous, event-driven controller interactions introduce new correctness challenges. In a control plane, correctness properties such as liveness are inherently non-local to each controller: they depend not only on a controller's own logic, but also on the interactions with other controllers. Figures 3(b) and (c) show two patterns of erroneous interactions. First, as shown in Figure 3(b), the Deployment controller's liveness is violated because its dependency on the ReplicaSet controller's liveness is unsatisfied. Second, unbounded interference between controllers can invalidate liveness, and this failure only occurs when controllers run together [57, 58]. In Figure 3(c), when the ReplicaSet object is repeatedly modified by other controllers, the two controllers fail to converge due to oscillation.

These challenges expose a fundamental gap in existing controller verification techniques. Specifically, Anvil [75] formalizes a general specification of individual controllers, known as Eventually Stable Reconciliation (ESR), which states that a controller should *eventually* reconcile the cluster to a desired state (*progress*), and then *always* keep the cluster in the desired state (*stability*). However, Anvil's practice of verifying a controller's ESR in a hardcoded, eventually quiescent environment is fundamentally limited in that it cannot verify a set of interacting controllers in the cluster control plane. Specifically, as a single-controller specification focusing on liveness, ESR does not guarantee that the controller never interferes with others. Therefore, even if each controller individually satisfies ESR in the hardcoded environment, their combination may fail to ensure global progress and stability because neither ESR nor the environment accounts for interactions between the controllers.

In this paper, we address this gap by introducing new verification techniques for reasoning about *collections* of interacting controllers in a *compositional* way. Our key idea is to make inter-controller interference explicit and to develop compositional specifications that allow correctness proofs of individual controllers to be combined. This enables reasoning about control-plane behavior at the level of the entire system, while preserving modularity.

3 Compositional Reconciliation (CORE)

We formalize a general correctness specification for cluster control planes called *COmpositional REconciliation (CORE)*. CORE is designed to capture the essential functionality of cluster control planes in a general and compositional form. CORE states that a set of controllers S collectively reconcile the cluster state to eventually match the desired state stably. Proving CORE for S ensures the absence of erroneous interactions that invalidate global progress and stability of S. To be compositional, CORE uses rely-guarantee style temporal predicates to account for controller interactions: the CORE specification for S has a premise that other controllers do not interfere with S, and it ensures that S does not interfere with other controllers. This allows developers to reuse the proof of CORE for S to prove CORE for the combination of S and other controllers. This enables a practical way to progressively increase the verified portion of a cluster control plane by gradually verifying more controllers.

3.1 The CORE Specification

CORE is a general specification that describes the behavior of a set of controllers, formalized in TLA (Temporal Logic of Actions) [50]. TLA reasons about a system described as a state machine. TLA formulas are predicates over the system's execution traces, which are infinite sequences of system states. We first give an abstract form of the CORE specification and then explain how to concretize its arguments. The abstract CORE specification in eq. (1) below says that given a state machine modeling the cluster control plane (Model), for *all* possible executions of Model, two conditions hold: (1) the target controllers S successfully reconcile the cluster state (Success), assuming other controllers do not interfere with them (Rely) and that lower-layer controllers successfully reconcile (Depend); (2) the target controllers S do not interfere with other controllers (Guarantee). The five arguments are concretized for each target controller set.

$$\mathsf{CORE}_{\mathsf{Abs}}(\mathsf{Model}, \mathsf{Rely}, \mathsf{Depend}, \mathsf{Success}, \mathsf{Guarantee}) \triangleq$$
$$\mathsf{Model} \models ((\mathsf{Rely} \wedge \mathsf{Depend}) \Rightarrow \mathsf{Success}) \wedge \mathsf{Guarantee} \quad (1)$$

We interpret $\mathsf{CORE}_{\mathsf{Abs}}$ for different target controller sets. For example, if the target controller set contains only the Deployment controller (Figure 1), CORE states that the Deployment controller successfully reconciles the cluster state

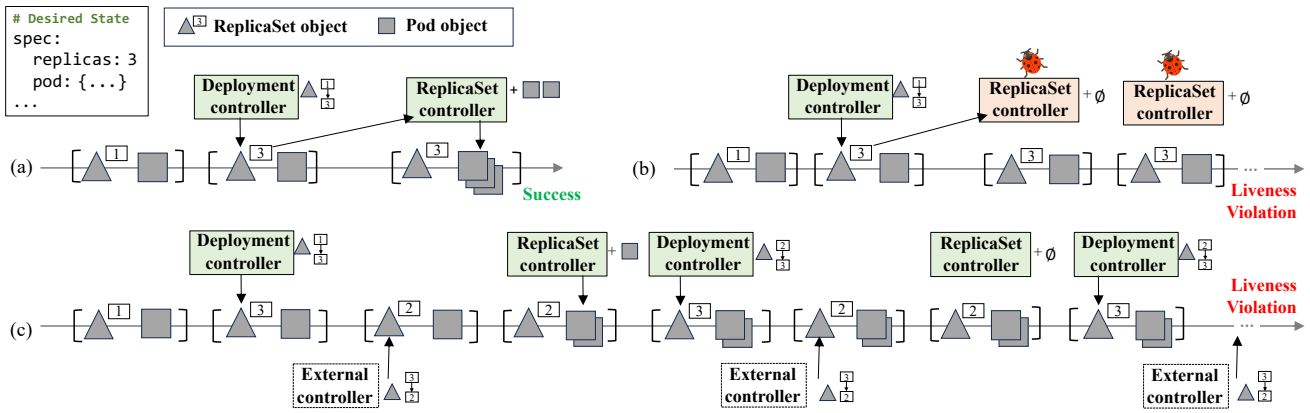

Figure 3. Controller interactions between the Deployment controller (Fig. 1) and the ReplicaSet controller (Fig. 2). (b): liveness violations due to unsatisfied liveness dependencies. (c): liveness violations due to interference (the ReplicaSet object is repeatedly modified).

if there is no interference from others (Rely) and the ReplicaSet controller (Figure 2) successfully reconciles the cluster state (Depend). CORE for just the Deployment controller is not enough: it would allow the reconciliation failures in Figure 3(b) and Figure 3(c). CORE for a target set consisting of the Deployment and the ReplicaSet controllers states that both of them successfully reconcile the cluster state if there is no interference from others, and CORE precludes failures in Figure 3(b). CORE also implicitly states that the two controllers do not interfere with each other so failures in Figure 3(c) do not happen between these two controllers. If S contains even more controllers, CORE would also state that there is no interference between any pair of these controllers.

CORE is designed so that its proofs are compositional. Combining proofs of CORE specifications of two controller sets S_1 and S_2 yields a stronger CORE specification for their composition $S_1 \cup S_2$. The Guarantee condition enables composition because it imposes a constraint on target controllers' behaviors; it forbids arbitrary actions that might interfere with others. The key to proving composition is to show the compatibility between S_1 and S_2's Rely and Guarantee. Compatibility means that if the behavior of S_1 (resp. S_2) satisfies its Guarantee, then it does not violate the Rely of S_2 (resp. S_1). If there is liveness dependency, e.g., S_1's progress depends on S_2's progress, then composition also requires proving that S_2's reconciliation Success implies S_1's Depend condition.

We concretize CORE's Success argument using ESR (Eventually Stable Reconciliation) [75], a general liveness property stating that a controller reconciles the cluster state to eventually match the desired state and then keeps the cluster state stable, assuming that the desired state eventually stabilizes. ESR of a lower-layer controller (e.g., ReplicaSet controller) can be used to concretize the Depend condition of an upper-layer controller (e.g., Deployment controller). A key challenge in defining CORE is formalizing the Rely and Guarantee conditions. We address this challenge in §3.2 and give the full definition of the concretized CORE in §3.3.

3.2 Defining Rely and Guarantee

CORE is inspired by rely-guarantee reasoning [46, 80] for concurrent multi-threaded programs. We adapt rely-guarantee reasoning to cluster control planes. First, state reconciliation is fundamentally a liveness property [75] which is expressed in temporal logic, so our rely-guarantee reasoning is formalized in temporal logic rather than a Hoare-style program logic as in prior work [90]. Second, we must address the challenge of defining the rely and guarantee conditions. These are typically hard to formulate because they are *global* properties; for a multi-threaded program, the rely condition for thread t needs to specify all possible effects made to the global program state by other threads between any two steps of t, and the guarantee condition for t needs to specify all possible effects to the global state made by each step of t. For cluster control planes, this would mean that whenever a new controller c joins the system, developers must revise the rely conditions of all the previously verified controllers to account for c's effects and then redo all the proofs.

We leverage two design patterns of cluster control planes to formalize the rely and guarantee conditions. First, controllers interact with each other by updating the shared cluster state, represented as state objects and exposed by API servers through state-centric interfaces [33, 74]. Each controller updates the cluster state by sending requests with clear semantics (e.g., creating, updating, and deleting state objects) to the API server. Second, controllers are designed to be loosely coupled microservices managing disjoint parts of the cluster state. Different controllers typically update distinct objects or disjoint parts of the same object. This is different from multithreaded programs where threads read-/write shared memory and directly interfere with each other.

Our solution is to design *request-centric* rely-guarantee conditions that specify a controller's assumptions on the effects of other controllers' requests (rely) and commitments about the effects of its own requests (guarantee). These conditions can be defined *locally* because of controllers' disjoint

reconciliation scopes. The conditions are formalized in eq. (2). The definition uses the temporal operator □ (always): a formula $\Box P$ asserts that P holds in the current state and all future states. The rely R is parameterized by a pair of controllers c and c' and specifies how c relies on c': it is always (□) the case that every (∀) request r sent by c' ($\texttt{sent}_{c'}(r)$) satisfies $\texttt{req\_rely}_c$. The guarantee G_c states that it is always the case that every request r sent by c satisfies $\texttt{req\_guarantee}_c$.

$$R_{c,c'} \triangleq \Box\big(\forall r,\ \texttt{sent}_{c'}(r) \Rightarrow \texttt{req\_rely}_c(r)\big) \tag{2a}$$

$$G_c \triangleq \Box\big(\forall r,\ \texttt{sent}_{c}(r) \Rightarrow \texttt{req\_guarantee}_c(r)\big) \tag{2b}$$

One might wonder whether we can relax $R_{c,c'}$ to a weaker assumption stating that c' might send arbitrary requests in the beginning but *eventually* stop doing that and afterwards only send requests that satisfy $\texttt{req\_rely}_c$. However, this assumption is too weak for some controllers. For example, the Kubernetes StatefulSet controller [25] uses a naming scheme for the Pods it manages. If another controller c' happens to use the same names for its Pods, the StatefulSet controller will never be able to create its Pods due to the naming conflict. Moreover, even after c' stops using those names to create Pods, the StatefulSet controller remains blocked because the Pods using the names still exist. Our definition of $R_{c,c'}$, with the leading □, is generally applicable to diverse controllers.

We present a general approach for defining `req_rely` and `req_guarantee` for Kubernetes controllers. Our approach leverages Kubernetes' *parent reference* mechanism [22] that defines controllers' ownership of the state objects. The parent reference is a field in each object o that points to another object that logically owns o, and Kubernetes controllers use this field to identify the objects they own. For example, the Deployment controller owns every ReplicaSet object whose parent reference points to a Deployment object. For a controller c, $\texttt{req\_rely}_c(r)$ requires that the request (sent by another controller) does not create, update, or delete any object owned by c, and $\texttt{req\_guarantee}_c(r)$ requires that the request (sent by c) only creates, updates, or deletes objects owned by c. A subtle corner case is ownership transfer: a controller can update an object's parent reference to adopt an orphan (an object with no parent reference) or release an owned object. The $\texttt{req\_rely}_c(r)$ further prevents other controllers from transferring ownership to c, and $\texttt{req\_guarantee}_c(r)$ prevents c from transferring ownership to other controllers. This is important because if another controller repeatedly assigns orphan ReplicaSet objects to the Deployment controller, the Deployment controller must scale each newly owned ReplicaSet, disrupting its progress.

As an example, Figure 4 shows part of the Deployment controller's `req_rely`, which prevents any update request from modifying a ReplicaSet object owned by a Deployment (line 9) or assigning a ReplicaSet's parent reference to a Deployment (line 10). The rely condition still allows other controllers to update ReplicaSet objects, provided they check the

```
1  spec fn req_rely(r: Request) -> StatePredicate {
2    |s : ClusterState| {
3      match r {
4        UpdateReplicaSetRequest(rs) => {
5          let result = s.find_in_store(rs.key());
6          result is Some ==> {
7            let cur_rs = result.unwrap();
8            cur_rs.version == rs.version ==>
9              ! cur_rs.parent() is Deployment
10             && ! rs.parent() is Deployment
11         }
12       }
13       ... // cases for other requests are omitted
14 }}}
```

Figure 4. The Deployment controller's `req_rely`. This function returns a closure (called a state predicate) that takes a cluster state and returns a bool. This style allows us to use the predicate in a temporal formula (e.g., after the □ operator). The update request `r` carries a new ReplicaSet object `rs`, and the update happens only if the object currently exists (line 6) and its version number equals the version number of incoming `rs` (line 8). The state predicate returns true if neither the current nor the incoming ReplicaSet object's parent reference is a Deployment object.

parent reference beforehand—a common and recommended practice in mature Kubernetes controllers.

CORE's request-centric rely-guarantee conditions apply to interactions beyond those involving parent references. For example, the StatefulSet controller's rely condition additionally requires that other controllers not issue requests to create Pods with names reserved by the StatefulSet controller's naming scheme.

An alternative to rely-guarantee reasoning is separation logic [47, 66], which leverages a notion of ownership to partition program state for compositionality. However, most automated verifiers [51, 52, 54] do not support separation logic, and verifying liveness with separation logic remains a challenge [73, 79]. We adopt rely-guarantee reasoning in this paper and leave using separation logic as future work.

3.3 Concretizing the CORE Specification

We concretize CORE_{Abs} (eq. (1)) for a controller set S using the request-centric rely-guarantee conditions and ESR, yielding $CORE_S \triangleq \text{CORE}_{\text{Abs}}(M_S, R_S, D_S, ESR_S, G_S)$. The expanded definition of $CORE_S$ is in eq. (3).

$$M_S \models ((R_S \wedge D_S) \Rightarrow ESR_S) \wedge G_S \tag{3}$$

$$R_S \triangleq \forall S' \perp S,\ \bigwedge_{\substack{c\in S\\ c'\in S'}} R_{c,c'} \qquad ESR_S \triangleq \bigwedge_{c\in S} ESR_c \qquad G_S \triangleq \bigwedge_{c\in S} G_c$$

The formalization of $CORE_S$ is a key contribution of this paper. The state machine M_S captures *all* possible behaviors of the cluster control plane, including the controllers $c \in S$, API servers, the datastore, and the network, under *all* possible control-plane configurations in which S coexists with arbitrary other controllers. R_S is the conjunction of $R_{c,c'}$ (eq. (2a)) for each c in S and c' in a disjoint set S' (denoted $S' \perp S$). Similarly, ESR_S and G_S are conjunctions of ESR_c

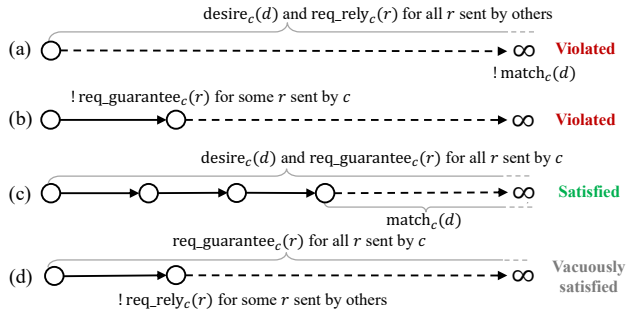

Figure 5. Executions that violate or satisfy the CORE specification of a controller c. (a) violates ESR because the cluster state never matches d while the rely condition is satisfied; (b) violates the guarantee condition (c at some point sends a request that violates $\texttt{req\_guarantee}_c$); (c) satisfies CORE: the cluster state eventually matches and always matches d and every request sent by c satisfies $\texttt{req\_guarantee}_c$; (d) vacuously satisfies CORE: the guarantee condition is satisfied but the rely condition is violated by another controller that sends some request violating $\texttt{req\_rely}_c$.

and G_c (eq. (2b)) over S, respectively. The variable D_S is the conjunction of any liveness dependencies of S on controllers outside S. If S is the singleton set of the ReplicaSet controller whose liveness does not depend on other controllers' liveness, $D_S = \mathit{True}$. If S is the Deployment controller, D_S is the ESR of the ReplicaSet controller. If S is the combination of both controllers, $D_S = \mathit{True}$ because the liveness dependency is resolved inside S.

ESR [75] captures progress and stability, the two key properties of state reconciliation, and is defined as follows.

$$ESR_c \triangleq \forall\, d,\ \Box\texttt{desire}_c(d) \rightsquigarrow \Box\texttt{match}_c(d). \tag{4}$$

ESR's definition uses another temporal operator $\rightsquigarrow$ (leads-to). A formula $P \rightsquigarrow Q$ states that at any point of the execution trace, if P holds, then *eventually* Q holds. ESR uses d to denote a state description, $\texttt{desire}(d)$ to denote that d is the current description of the desired state, and $\texttt{match}(d)$ to denote that the current cluster state matches d. ESR states that if at any point the desired state stops changing ($\Box\texttt{desire}_c(d)$), then the cluster will eventually ($\rightsquigarrow$) reach a state that matches it, and stay that way forever ($\Box\texttt{match}_c(d)$).

The CORE specification is a combination of liveness and safety properties. The liveness property ($(R_S \wedge D_S) \Rightarrow ESR_S$) states that, if controllers not in S do not modify the objects owned by S (R_S) and the liveness dependencies of S are resolved (D_S), then each controller in S eventually makes the cluster state match its desired state stably as long as its desired state remains unchanged (ESR_S). The safety property (G_S) states that controllers in S never modify objects that are not owned by them. This safety property is crucial for composition because it implies that controllers in S never modify objects owned by controllers in a disjoint set S', and vice versa. Figure 5 illustrates CORE for a singleton controller set $\{c\}$ with no liveness dependency on other controllers.

An important design decision of CORE is that it includes the conjunction of ESR of *all* controllers in S, instead of only exposing ESR of a few top-level controllers. Specifically, if there is some top-level controller c that a client relies on, then CORE for a set S that includes c implies ESR_c, so the client can readily use that; but importantly CORE also encodes the assumptions (e.g., non-interference) needed by "internal" controllers that the top-level controller depends on. Moreover, by conjoining ESR for all controllers we allow the set of controllers considered to be increased incrementally.

The practical implication of CORE. The CORE specification provides a strong correctness guarantee. ESR_S precludes liveness bugs with diverse root causes [75], and G_S precludes safety bugs where a controller incorrectly manages objects it does not own—a common class of bugs in mature controllers reported by prior work [58, 74] and in the field [7–9, 12]. Our analysis of all 188 bugs found by state-of-the-art controller testing tools [41, 42, 74] shows that CORE precludes 178 (95%) of them. These bugs have diverse root causes, including: (1) the controller fails to handle corner-case input, (2) the controller cannot recover from intermediate states caused by unexpected crashes, and (3) the controller takes wrong actions after seeing a stale state. CORE precludes them because these bugs either block the controller's progress indefinitely, violating CORE's liveness property (ESR_S), or cause the controller to mistakenly update or delete objects not owned by it, violating CORE's safety property (G_S).

CORE's compositionality enables a practical way to progressively increase the verified portion of a cluster control plane by gradually incorporating newly verified controllers. We explain CORE's composition rules in detail in §4.4.

4 Proving CORE with Welder

Welder provides a general approach for developers to prove the CORE specification for a set of controllers S in a compositional way. Developers first prove CORE for each controller in S separately, then compose the CORE of each controller to prove CORE for S. The key challenge is to prove ESR for a controller c in the presence of a set of open, *arbitrary* controllers sharing the cluster state. This is different from prior work [75] which uses a closed, fixed environment where every component except the target controller is hardcoded. In our case, developers must consider any interaction from other controllers, constrained only by c's rely condition and liveness dependency condition. The new proof setting poses four challenges; we outline these briefly as C1–C4, then describe Welder's solutions in the rest of the section.

C1. Reasoning about liveness dependencies. Resolving a controller's liveness dependency requires applying another controller's ESR. A common pattern is that an upper-layer controller c_U needs to *iteratively* update the desired-state description of a lower-layer controller c_L and wait for c_L's progress (e.g., in rolling updates). It might seem that ESR does

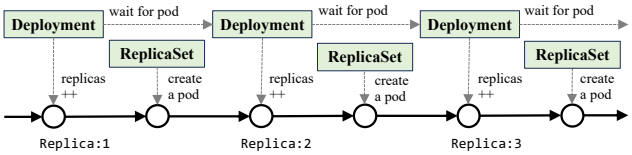

Figure 6. Rolling update process of the Deployment controller which gradually scales up a new ReplicaSet object for running the new version of an application.

not apply here because it states that if *the desired state description remains stable*, then eventually the cluster state matches the desired state stably. This creates a proof dilemma: proving c_U's progress requires applying c_L's ESR, which further requires c_U to stop updating c_L's desired state description.

C2. Reasoning about object sharing. Another common interaction pattern is multiple controllers updating disjoint parts of the same object to collectively update the cluster state. In version-based systems like Kubernetes, one would succeed and increment the version number, and others would fail. Proving liveness requires new fairness assumptions to preclude traces where one controller never wins the version race; traditional assumptions like weak or strong fairness are insufficient to prevent such starvations by version conflict.

C3. Reasoning about interference. Liveness proofs for progress are entangled with safety proofs for non-interference, such as proving that no request in the network will interfere with the controller's reconciliation. This poses a heavy proof burden because developers need to reason about effects of different types of requests (e.g., creation, update, and deletion) during the liveness proof.

C4. Composing proofs. After proving that each individual controller implements CORE, developers need to compose the proofs into CORE for the whole set. This requires developers to reason about whether the to-be-composed controllers might interfere with each other, and how the liveness dependencies (if any) are satisfied.

4.1 Proof Rule for Liveness Dependencies

A key challenge in proving a controller's CORE specification is reasoning about an upper-layer controller's liveness dependency on lower-layer controllers (**C1**). This requires proving the upper-layer controller's ESR_U by applying each lower-layer controller's ESR_L. Recall that ESR is defined as $\forall d,\ \Box\texttt{desire}_c(d) \rightsquigarrow \Box\texttt{match}_c(d)$ (eq. (4)). To prove that the cluster state stably matches the desired state ($\Box\texttt{match}_c(d)$), one must prove the stability of the desired state ($\Box\texttt{desire}_c(d)$). However, the upper-layer controller may *iteratively* update the lower-layer controller's desired state and wait for the lower-layer's reconciliation to complete after each update, a common pattern in controllers (e.g., [16–18, 20, 27, 30]). This creates a proof dilemma: proving ESR_U requires applying ESR_L, but the iterative updates "contradict" ESR_L's premise on desired state stability.

We use the Deployment controller's rolling update feature, which has a liveness dependency on the ReplicaSet controller, to illustrate the problem. The Deployment controller updates the ReplicaSet object, which is the desired state description of the ReplicaSet controller. As shown in Figure 6, instead of directly setting the ReplicaSet object's `replicas` value to the application's desired replicas count, the Deployment controller iteratively increases `replicas` and waits after each increase for the ReplicaSet controller to reconcile. When proving the Deployment controller's ESR, developers use the ReplicaSet controller's ESR and do not need to know the implementation of the ReplicaSet controller. However, how can we satisfy the premise of the ReplicaSet controller's ESR, which requires the stability of `replicas`, while the Deployment controller iteratively updates `replicas`?

Our insight is that the stability of `replicas` depends on the termination of the Deployment controller's iterative update. The Deployment controller will stop increasing the value of `replicas` once it reaches the desired replica count, and then `replicas` will be stable afterwards.

The next question is how to prove the termination of the iterative update. Note that the Deployment controller's update is *monotonic*—it only increases `replicas`—which is the key to the termination. We can prove that `replicas` eventually becomes stable. We denote the difference between the desired replicas and the ReplicaSet's `replicas` as x, and define $S(n)$ to state that if $x = n$, then `replicas` eventually becomes stable. We prove $S(n)$ for any n by strong induction.

1. Base case. We prove $S(0)$: the value of `replicas` already equals the desired replicas count, so the Deployment controller stops its update.
2. Inductive step. We prove $S(k)$ for any $k > 0$. Two cases:
   a. If the Deployment controller updates `replicas` again at some point, then the update must increase `replicas`, thus decreasing x to k' where $k' < k$. By the inductive hypothesis $S(k')$ where $k' < k$, we prove $S(k)$. Intuitively, this is how x gets closer to 0 step by step.
   b. The Deployment controller no longer updates `replicas` from now on, which makes `replicas` stable.

This proves $S(n)$ for any starting n, so the Deployment and ReplicaSet controllers eventually reconcile the `replicas`.

We develop a new proof rule, Iter-ESR, to generalize the reasoning pattern for applying ESR with iterative update:

$$\frac{\begin{array}{l}\forall n \in \mathbb{N}, M \models \Box P(n) \rightsquigarrow \Box Q(n) \\ \forall n \in \mathbb{N}, M \models \Box\big(P(n) \Rightarrow \Box\big(\exists m, m \le n \wedge P(m)\big)\big) \\ \forall n \in \mathbb{N}, n > 0 \Rightarrow \big(M \models \Box Q(n) \rightsquigarrow \neg P(n)\big)\end{array}}{\forall n \in \mathbb{N}, M \models P(n) \rightsquigarrow \Box P(0)}\ \textsc{Iter-ESR}$$

The key insight behind Iter-ESR is that an upper-layer controller can be proved to make progress without establishing the stability of every intermediate version of the lower-layer controller's desired state. The rule applies to a

common interaction pattern: an upper-layer controller iteratively updates the desired state of a lower-layer controller and, after each update, waits for the lower-layer controller to make progress. We explain it below using the interaction between the Deployment and ReplicaSet controllers.

The rule's first premise is a different form of ESR parameterized by a natural number n. In the rolling update example, we rewrite the ReplicaSet controller's ESR in this form. $P(n)$ states that the difference between the Deployment's desired replicas and the `replicas` of the ReplicaSet object rs is n. $Q(n)$ states that the difference between the Deployment's desired replicas and the number of Pods (created by the ReplicaSet controller) is n, meaning that the cluster state matches rs. The usage of n is similar to the inductive proof above.

The second premise states that it is always true ($\Box$) that, if $P(n)$ holds at some point, then $P(m)$ holds afterward with an $m \leq n$, i.e., the monotonicity property discussed above. It means that the difference between the desired replicas and `replicas` either remains unchanged (when the Deployment controller is waiting) or gets decreased (when the Deployment controller increases `replicas`).

The third premise states that if $n > 0$ and $Q(n)$ keeps ($\Box$) holding true, then eventually ($\rightsquigarrow$) $P(n)$ does not ($\neg$) hold true. It means that if the desired replicas number is not reached yet ($n > 0$), and all the Pods requested by the Deployment controller have been created, the Deployment controller eventually stops waiting and increases `replicas`, thus invalidating $P(n)$—the difference is no longer n.

Finally, the conclusion states that starting from $P(n)$ for any n, eventually $P(0)$ holds and keeps holding afterwards, e.g., the ReplicaSet object's `replicas` eventually equals the desired replicas, and becomes stable. This conclusion implies stability of the lower-layer controller's desired state description, which enables the application of the lower-layer controller's ESR. We have mechanized the Iter-ESR rule in Verus, proved its soundness, and used it to verify the Deployment controller's rolling update feature.

4.2 Reasoning about Object Sharing

Controllers often share an object and update disjoint parts of the object as a way to collectively update the cluster state. For example, the Deployment controller updates a ReplicaSet object's `spec` field while the ReplicaSet controller updates the `status` field. This leads to liveness issues in version-based systems where each object has a version number and each update must carry the latest version number to succeed—a common concurrency control mechanism in datastore systems. When multiple controllers update the same object at the latest version, one would succeed and increment the version number, while others would fail due to version conflict; this is common in cluster control planes using version-based datastores (e.g., etcd in Kubernetes and ZooKeeper in Twine). Object sharing leads to a liveness issue: if some controller always loses the race then it cannot make progress (**C2**). The trace where one controller always loses is unrealistic: in practice if a controller wins the race it will not immediately issue the next update, giving the other controller enough time to retry its update with the new version number. However, this trace is allowed by our generic environment model that considers all possible event interleaving.

The common solution to preclude such unrealistic traces is to incorporate *fairness assumptions* that preclude environmental unfairness into the liveness proof. However, traditional fairness assumptions such as TLA's weak fairness and strong fairness [50] do not preclude unfair competitions. The reason is that both weak and strong fairness state that under some condition an action A eventually happens. In our environment model, A is the action that the API server handles the controller's update request. The unfair competition is not caused by A never happening—it is caused by A always happening *after* the API server handles another update request. Anvil [75] addresses the problem by assuming that all other (hardcoded) controllers eventually stop updating the object shared with the target controller until the target controller updates the object. This assumption is unrealistic when other controllers need to periodically update the object and is not compositional because it favors one controller by blocking other controllers' progress. Anvil's assumption also complicates the liveness proof because developers need to consider two cases in the proof: other controllers can still update the object, or they have stopped.

Welder introduces a transactional abstraction to update without version conflicts and an implementation that simulates this abstraction using existing APIs, and justifies the abstraction based on an appropriate fairness assumption. Ideally, a controller should use a transactional "check-then-update" API that atomically (1) reads an object, (2) checks some condition on the object if the object exists, and (3) updates the object using the version number from the read if the condition holds. The atomicity guarantees the absence of version conflicts. However, conflict-free transactions are not always available in practice. For example, despite proposals on transaction support [1, 48], Kubernetes does not provide a transactional API. Our insight is that the conflict-free transaction can be simulated using a *Retry-on-Conflict (RoC) loop* with existing APIs as shown in Figure 7. The essence is that the client of the RoC loop never sees a version conflict. The RoC loop is general: the `cond` can be any condition check on `obj` (e.g., checking the parent reference of `obj`), and the `new_obj` can be constructed based on `obj` (e.g., increasing the `replicas` field of `obj`).

Our fairness assumption states that the RoC loop terminates, which enables liveness verification in version-based systems. It precludes unfair competitions: if the RoC loop terminates, it means that the client of the RoC loop either eventually wins the version competition, or it encounters other errors (e.g., the object does not exist or the `cond` does not hold). The assumption is realistic. The loop terminates in

```
loop {
  match get(key) {
    Err(err) => return Err(err),
    Ok(obj) => { // obj carries the version number
      if !cond(obj) { return Err(...); }
      // new_obj carries the same version of obj
      let new_obj = ...;
      match update(key, new_obj) {
        Err(VersionConflict) => continue,
        Err(err) => return Err(err),
        Ok(o) => return Ok(o),
      }
    }
  }
}
```

Figure 7. The Retry-on-Conflict (RoC) loop that simulates the transactional semantics. The code retries on a version conflict. If the code terminates, it simulates a transaction that atomically gets the object and updates the object conditionally. The version conflicts never surface to the client of this code.

practice because the controller uses the version number from the most recent `get` in its `update`. The assumption is also easy to use in liveness proofs: under this assumption, the RoC loop refines the transactional "check-then-update" API (the two are externally indistinguishable). The refinement means that if developers prove liveness of a controller model that uses transactional APIs, then it implies liveness of a controller implementation that uses RoC-loop based APIs. The liveness proof enjoys the simple transactional semantics and does not need to reason about any version conflict.

We prove that the RoC loop in Figure 7, denoted as s (source), refines the transactional "check-then-update" API, denoted as t (target). We prove that every behavior of s can be mapped to a behavior of t in terms of the cluster state under our fairness assumption. If s terminates after n iterations, then the previous $n-1$ iterations make no change to the cluster state so they are mapped to a no-op in the trace of t before the transaction happens. For the nth iteration, we consider two cases. First, if s returns an error before `update`, then the trace of s can be mapped to the trace of t that reads the cluster state at the same point as the `get` of s and returns with the same error. Second, if s returns after `update`, then the trace of s can be mapped to the trace of t that atomically reads and updates the cluster state at the same execution point. If the `update` succeeds, it implies that no other request modifies the object between the `get` and `update` of s, so s appears transactional just like t; otherwise neither s nor t changes the cluster state, and they return the same error.

Welder provides RoC-loop based APIs for implementing controllers, and the corresponding transactional APIs in its environment model for verifying controllers.

4.3 Proof Strategy for (Non-)Interference

When proving that a controller eventually reconciles the cluster state to match the desired state description d, developers need to reason about two types of interference: external interference from other controllers and internal interference

```
proof fn lemma_state_preservation(m, s, s', c, d, req)
requires
  m.next_step(s, s', HandleReqStep(req)),
  ! req_sent_by(req, c, d),
  ... // other requires (e.g., rely) are omitted
ensures preserved(d, s, s')
{ ... } // proof body is omitted
```

Figure 8. The lemma for state-preservation invariant. The lemma states that if (1) the state machine model `m` transitions from state `s` to state `s'` by taking a step for the API server to handle the request `req`, and (2) the `req` is not sent by controller `c` for desired state description `d`, then the cluster state managed by `d` is preserved from `s` to `s'`. Some details are omitted for clarity.

from the same controller's other reconciliations for other desired state descriptions d' (**C3**). For example, to prove that an update request u sent by the Deployment controller successfully updates a ReplicaSet object, developers need to prove (1) the absence of external interference by showing that requests sent by other controllers will not overwrite u's change on the ReplicaSet object or delete the ReplicaSet object, and (2) the absence of internal interference by showing that the Deployment controller's different reconciliations never share the same ReplicaSet object. This poses a heavy proof burden because developers need to reason about effects of different types of requests (e.g., creation, update, and deletion) from different controllers during the liveness proof.

To reduce the proof burden, Welder provides a new proof strategy based on a unified state-preservation invariant. Instead of enumerating effects of different requests, the state-preservation invariant simply states that the cluster state managed by the target controller's reconciliation for desired state description d is preserved by any request that is not from the target controller's reconciliation for d. We define the state-preservation invariant as a lemma in Figure 8. Our state-preservation invariant corresponds to the stable assertion in classic rely-guarantee reasoning [46, 82]: assertions on the state managed by d are stable under any interference allowed by the rely condition.

To use the state-preservation invariant, developers need to define what part of the cluster state is managed by `d`. Take the Deployment controller as an example: its desired state description `d` is encoded by a Deployment object, and the state managed by `d` includes the ReplicaSet objects whose parent reference (§3.2) points to the Deployment object.

The state-preservation invariant improves modularity of the proof by cleanly dividing the ESR proof into two parts. First, developers prove this invariant by applying the rely condition on other controllers' requests and reasoning about the target controller's implementation. Second, developers apply this invariant when reasoning about the progress of the target controller. Developers prove that each step taken by the target controller `c` for its desired state description `d` makes the cluster state closer to `d`, while each other step preserves the state for `d`. We have used this proof strategy to prove ESR for four Kubernetes controllers.

4.4 CORE Composition

Welder provides proof rules to automate the composition of CORE (**C4**). To explain the composition rule, we introduce two new notations. First, we define R_{S_1,S_2} as $\bigwedge_{c\in S_1, c'\in S_2\setminus S_1} R_{c,c'}$, meaning S_1's rely assumption on S_2. We use $S_2 \setminus S_1$ in the definition because our composition rules allow composing two overlapping controller sets. Second, we define $CORE^{\star}_{S}$ as $CORE_S \wedge (D_S = True)$, meaning that $CORE_S$ holds and S has no liveness dependency on other controllers.

Welder provides the Compose rule (eq. (5)) to compose controller sets without liveness dependencies. To apply the Compose rule, developers need to prove that S_1 and S_2's rely and guarantee conditions are *compatible*, denoted as $Compat_{S_1,S_2}$. The definition says that G_{S_1} implies R_{S_2,S_1}, and G_{S_2} implies R_{S_1,S_2}. The Compose rule proves $CORE^{\star}_{S_1\cup S_2}$ given $CORE^{\star}_{S_1}$, $CORE^{\star}_{S_2}$, and $Compat_{S_1,S_2}$.

$$\frac{CORE^{\star}_{S_1} \qquad CORE^{\star}_{S_2} \qquad Compat_{S_1,S_2}}{CORE^{\star}_{S_1\cup S_2}}\ \textsc{Compose} \quad (5)$$

$$Compat_{S_1,S_2} \triangleq M_{S_1\cup S_2} \models (G_{S_1} \Rightarrow R_{S_2,S_1}) \wedge (G_{S_2} \Rightarrow R_{S_1,S_2})$$

The intuition of Compose is that $Compat_{S_1,S_2}$ proves no interference between S_1 and S_2. We find proving $Compat_{S_1,S_2}$ straightforward because the proof can be reduced to reasoning about the requests: for any $c_1 \in S_1$ and $c_2 \in S_2 \setminus S_1$, if a request r satisfies $\texttt{req\_guarantee}_{c_1}$ (resp. $\texttt{req\_guarantee}_{c_2}$) then it satisfies $\texttt{req\_rely}_{c_2}$ (resp. $\texttt{req\_rely}_{c_1}$).

Welder also provides Compose-Dep to compose S_1 and S_2 when S_2 depends on S_1's liveness (eq. (6)). Compose-Dep requires $ESR_{S_1} \Rightarrow D_{S_2}$, which resolves S_2's liveness dependency, so the composition $S_1 \cup S_2$ has no liveness dependency. Often, D_{S_2} equals ESR_{S_1}, so the implication holds trivially.

$$\frac{CORE^{\star}_{S_1} \qquad CORE_{S_2} \qquad Compat_{S_1,S_2} \qquad M_{S_1\cup S_2} \models ESR_{S_1} \Rightarrow D_{S_2}}{CORE^{\star}_{S_1\cup S_2}}\ \textsc{Compose-Dep} \quad (6)$$

CORE and its composition rules enable a practical way to progressively increase the verified portion of a cluster control plane by gradually incorporating newly verified controllers.

5 The Welder Framework

We implement Welder as a framework on top of Anvil [75] and Verus [51, 52]. The proof techniques in §4 are mechanized as lemmas and models in the Welder framework. With Welder, developers write controller implementations as state machines in Rust and proofs for the implementations and their compositions in Verus. Welder reuses the TLA embedding from Anvil so that developers can write specifications and proofs using temporal operators, e.g., $\Box$ and $\rightsquigarrow$.

Parametric environment model. To support compositional verification, Welder provides a state machine model that describes the behavior of components that controllers interact with, including the network and the API server. The model is parameterized by a set of controllers, reflecting that cluster control planes can be configured with different controllers. Welder allows developers to instantiate the model using a specific set of controllers S and arbitrary other controllers when verifying that S implements CORE—this is the model M_S in eq. (3). This instantiated model captures all possible interactions between S and arbitrary other controllers. This differs from Anvil's environment model that hardcodes all the components except the single target controller.

Simulating variadic generics in Verus. For flexibility, Welder allows developers to define their controller's local state using any data type. However, this makes it challenging to instantiate the environment model with an arbitrary number of controllers, because Verus does not support variadic generics, so the environment model cannot take an arbitrary number of type parameters. To address this problem, Welder simulates variadic generics by using an `OpaqueT` type as a uniform representation of all controllers' local states in the environment model. Developers can still define each controller's local state using any data type `T`, but they need to axiomatize a pair of `encode: T -> OpaqueT` and `decode: OpaqueT -> Option<T>` operations to convert between the concrete and opaque types. The operations `encode` and `decode` are uninterpreted functions (i.e., they have no function bodies) and are related using an axiom that for any `t: T`, `decode(encode(t)) = Some(t)`. Developers apply this axiom when reasoning about their controller's local state.

Environment assumptions. Welder assumes an asynchronous environment: there are no finite upper bounds on message-delivery delays or computation time. To prove liveness, Welder uses TLA's weak fairness assumption [50]: if an action remains enabled, it will eventually happen. For example, it implies that messages (if not dropped) will eventually get delivered. Welder inherits Anvil's fault model, which includes message drops and controller crashes, and a liveness assumption that allows faults to occur arbitrarily many times but requires them to eventually stop happening.

Modeling Pod failures. Welder extends Anvil's fault model to further model Pod failures. This is important because Welder supports verifying both custom controllers (Anvil's focus) and core controllers, and many core controllers such as the ReplicaSet and StatefulSet controllers directly manage Pods. Core controllers should tolerate failures of the Pods they manage, e.g., if a Pod is deleted due to a container crash, the ReplicaSet controller should create a new Pod. Welder models Pod failures in a simple and general way by having a PodMonkey that nondeterministically creates, deletes, or updates Pods. To enable liveness verification, Welder has a liveness assumption that the PodMonkey action can happen an arbitrary number of times but eventually stops. Note that the StatefulSet controller requires a stronger assumption that the PodMonkey always respects its naming scheme (§3.2).

Trusted code. Welder relies on the following components: (1) the TLA embedding from Anvil, (2) the parametric environment model, (3) the Kubernetes client library controllers use to query and update the cluster state, (4) the Verus verifier, the Rust compiler, and the underlying operating system.

6 Experience

We have used Welder to build and verify a fraction of the Kubernetes control plane—three *core controllers* for ReplicaSet [23], Deployment [20], and StatefulSet [25], and a custom controller for managing RabbitMQ (a message-broker application) [29]. We proved that the composition of the four controllers implements the CORE specification. The three core controllers are implemented based on the official Kubernetes implementation [21, 24, 26], and we ported the RabbitMQ controller from Anvil to Welder. All the verified controllers can be directly deployed in real-world Kubernetes platforms. We found and fixed seven bugs in our controller implementations that violate CORE.

Feature differences. Our verified controllers implement key features of their reference implementations but do not provide full feature parity. The Deployment and StatefulSet controllers support features including scaling, version upgrades, and rolling updates. The ReplicaSet controller supports all features except Pod adoption and release and batched Pod creation and deletion. The ported RabbitMQ controller provides the same features as the one from Anvil.

New challenges of verifying core controllers. Core controllers require different specification and proof patterns from custom controllers (the focus of prior work [75]). Core controllers typically manage object types with *arbitrary* numbers of instances, while custom controllers manage a *fixed* number of instances per type. The reason is that a core controller provides primitives for custom controllers to manage applications; a custom controller orchestrates different object types to implement application-specific reconciliation, e.g., the RabbitMQ controller uses a StatefulSet object to manage RabbitMQ nodes, a ConfigMap object for configuration of each node, and a Secret object for authentication.

In core controllers, the names of objects they manage are often dynamically assigned, e.g., the ReplicaSet controller lets the API server randomly generate names of Pods it creates. So, we cannot refer to objects by name in our proofs. We used existential quantifiers (∃) to define ESR of core controllers, e.g., for the ReplicaSet controller, the `match` predicate states that there exists (∃) a set of Pods in etcd such that each Pod satisfies certain conditions and the set cardinality is the desired replica count. (The RabbitMQ controller's `match` simply states that a StatefulSet object with a particular name exists in etcd and its content satisfies the condition.) To prove ESR for the ReplicaSet controller, we proved that the set of Pods eventually reaches the desired replica count.

Porting the RabbitMQ controller from Anvil to Welder. Welder inherits Anvil's API for implementing controllers as state machines so controller code needs no change. The main effort was to prove ESR in Welder's parametric environment model, assuming its rely condition. The new ESR proof supports compositional verification but is harder, because Welder's environment model (for compositionality) allows arbitrary interactions between the RabbitMQ and other controllers, and we need to prove non-interference using the rely condition. We followed the proof strategy using the state-preservation invariant in §4.3 to reduce the proof burden. It takes about 30% more lines of proof for the RabbitMQ controller's ESR in Welder compared to the proof in Anvil.

Controller composition. After proving that each controller implements CORE, we used Compose-Dep (eq. (6)) to prove CORE for two pairs of controllers independently: (1) the ReplicaSet and Deployment controllers, and (2) the StatefulSet and RabbitMQ controllers. We then used Compose (eq. (5)) to compose all four controllers. Within each pair, there is a liveness dependency, but the two pairs do not depend on each other. Our composition order reflects how developers can compose controllers: first compose the stack of controllers with liveness dependencies, and then compose the controller stack with other independent stacks.

Bugs precluded by verification. Through verification with Welder, we found and fixed seven bugs in our controllers. One bug was inherited from the official StatefulSet controller, which creates a naming collision that confuses the ReplicaSet and StatefulSet controllers into competing for the ownership of the same Pod [28]; a similar bug was reported by others and confirmed by Kubernetes developers [2]. Six of the found bugs cannot be detected by existing controller testing tools [41, 42, 74]; they either require complex triggering conditions beyond the testing tools' exploration, or manifest silent failures that cannot be caught by the testing oracles.

7 Evaluation

Our evaluation shows that Welder enables pragmatic compositional verification of controllers (§7.1). The verified controllers pass extensive testing (§7.2) and achieve competitive performance compared to unverified references (§7.3).

7.1 Verification Effort

Verifying the four controllers took 1.5 person-years. We spent about one year building Welder and verifying the ReplicaSet and Deployment controllers. With this experience, we verified other controllers faster (e.g., 40 days for RabbitMQ). Table 1 shows the size of the controller proofs. The proof includes two parts. First, we prove that each controller implementation conforms to a controller model. The controller model is a mathematical representation of the executable controller code to enable TLA-style reasoning. The conformance proof is straightforward and takes less than 400 lines

Controllers	Trusted		Exec.	Proof		
	Spec.	Unver.		Model	Conf.	CORE (ESR)
ReplicaSet	205	257	300	418	119	7726 (7305)
Deployment	254	387	365	563	190	10327 (10201)
StatefulSet	194	380	896	1274	366	12085 (11937)
RabbitMQ	247	298	1619	1660	175	6920 (6772)
Composition	181	0	0	0	0	429 (0)
Total	1081	1322	3180	3915	850	37487 (36215)

Table 1. Code sizes of the controllers verified using Welder. **Trusted** includes the verified CORE specification and the unverified components. **Proof** includes (1) a controller model, (2) a conformance (**Conf.**) proof that the implementation (**Exec**) conforms to the model, and (3) the CORE proof, with the ESR proof in brackets.

per controller. Second, we prove that each controller model, when running in Welder's parametric environment model, satisfies the CORE specification. More than 94% of each controller's CORE proof is about ESR. In contrast, proving each controller's guarantee condition by induction takes less than 450 lines. Checking proofs of all controllers takes about 7 minutes on a laptop with 8 cores and 16GB memory.

The proof-to-exec ratio ranges from 5.4 to 30.4 across four controllers. The Deployment controller has the highest proof-to-exec ratio: verifying its rolling update feature requires extra proof effort (§4.1). The RabbitMQ controller, despite having more implementation code, requires the lowest proof-to-exec ratio due to its simple implementation logic: it follows the same "get-then-create-or-update" workflow for all object types, allowing us to reuse the same lemma across them. Proving ESR generally requires more effort than in Anvil, because we need to apply each controller's rely condition during the proof to rule out interference. We follow Welder's proof strategy to reduce the proof effort (§4.3).

The composition proof is lightweight, requiring only 429 lines to compose all four controllers. We apply Welder's composition rules (§4.4), and most of the effort lies in proving compatibility of controllers' rely and guarantee conditions. This demonstrates CORE's compositionality: once a controller is verified against its CORE specification, composing it with other controllers requires low additional proof effort.

Our verified controllers have relatively large unverified components. This is because they depend on the unverified Kubernetes client library `kube-rs` [19], and we use `kube-rs` to define the Kubernetes objects used as desired state descriptions. To enable verification in Verus, we define wrapper types with trusted (unverified) getter and setter methods to integrate objects defined using `kube-rs`. We expect this to shrink once Verus supports dual-mode functions [15].

7.2 Controller Correctness

We run extensive functional and crash tests on the verified controllers to validate their correctness using state-of-the-art controller testing tools [41, 42, 74]. All four controllers are deployed on Kubernetes to test their interactions.

Controller	Functional tests	Controller crash		Pod crash
		Individual	Correlated	
Deployment + ReplicaSet	465	552	276	401
RabbitMQ + StatefulSet	164	316	158	276

Table 2. Testing effort of the four verified controllers. No bugs were found during testing.

Controller	`reconcile` (sec.)			End-to-end (sec.)		
	Verified	Ref.	Diff	Verified	Ref.	Diff
ReplicaSet	0.85	0.12	0.73±0.33	8.95	8.83	0.11±0.60
Deployment	0.12	0.04	0.08±0.05	9.07	8.88	0.19±0.59
StatefulSet	0.50	0.32	0.18±0.10	121.93	123.56	-1.63±7.90
RabbitMQ	0.54	0.92	-0.39±0.78	122.46	124.48	-2.02±8.00

Table 3. Comparison of controllers' `reconcile` execution time and end-to-end system reconciliation time between the verified controllers and their references. The Diff column shows the difference (verified − reference) with standard deviation.

We run three types of tests: (1) 629 functional tests that exercise different features of each controller using diverse desired state descriptions generated by Acto [41, 42], (2) 1302 controller-crash tests using Sieve's approach [74] to check if the controllers can recover from crashes during their reconciliation, including 434 correlated crash tests that crash a pair of interacting controllers simultaneously, and (3) 677 Pod-crash tests that crash Pods managed by controllers to check if the controllers can recover the cluster state. Table 2 presents the number of tests. No bug was found.

7.3 Controller Performance

For the features they share (§6), the verified controllers achieve end-to-end performance comparable to that of their references in the official Kubernetes implementation. For each functional test (§7.2), we measure (1) execution time of each controller's `reconcile` function, and (2) the time it takes for the system to be fully reconciled. For each controller, the tests exercise only the features supported by both the verified controller and its reference. The experiments are run on CloudLab Clemson c6420 machines with dual Xeon Gold 6142 processors, 384GB DRAM, and a 6Gb/s HDD.

As shown in Table 3, the verified core controllers' `reconcile` execution times are modestly higher than those of the references, while the verified RabbitMQ controller is faster than its reference; the result is consistent with Anvil's evaluation because the verified RabbitMQ controller does not implement all the features from its reference. On the other hand, the end-to-end differences are negligible across all controllers, falling within a standard deviation. This is because controllers' `reconcile` makes up only a small fraction of the end-to-end system reconciliation time, which is dominated by factors outside the controllers' control, such as container startup and image pulling. None of the controllers is resource intensive. During experiments, the CPU consumption is less than 3% for both verified and reference controllers.

We also evaluate if the verified controllers introduce more load on the etcd datastore which is often the bottleneck for Kubernetes scalability [31, 76]. We measure the CPU and memory consumption of etcd and the Kubernetes API server, as well as etcd's disk I/O. The verified controllers do not add significant load to either etcd or the API servers. The differences in all metrics are within one standard deviation.

8 Related Work

Anvil. The closest related work is Anvil [75]. Anvil focuses on single-controller correctness by verifying ESR in a hard-coded environment, whereas Welder targets control-plane correctness through compositional verification of multiple controllers. Welder's CORE specification uses request-centric rely-guarantee conditions and Anvil's ESR to specify controller liveness in a compositional way. Welder also introduces new verification techniques to reason about controller interactions, including controller liveness dependencies.

Liveness verification for systems. Most systems verification efforts so far focus on safety rather than liveness [34–40, 43, 44, 49, 53, 55, 59, 65, 71, 72, 78, 83, 85, 86, 88, 89]. Prior work has verified liveness for other distributed systems that implement Paxos [45] and BFT [56, 70, 87], and a concurrent filesystem [90]. Welder targets verifying cluster control planes, a different type of system with different liveness properties. Unlike previously verified liveness properties (e.g., eventual response or filesystem termination), Welder's CORE specification captures the essence of state reconciliation and supports open composition.

Prior work has developed automated liveness verification techniques for system protocols. LVR [84] proves liveness of distributed protocols by synthesizing ranking functions. Ivy [63, 69] proves liveness of distributed protocols using a decidable fragment of first-order logic [67, 68], a more restricted modeling logic compared to the logic Welder uses. Alloy [6, 32, 60] supports modeling liveness properties but can only check finite instances. Compared to Welder, these techniques achieve a higher degree of proof automation, but they focus on verifying system *protocols*, not *executable implementations*. We are exploring the potential to incorporate these techniques to reduce the proof effort in Welder.

McMillan [62] proposed a compositional approach for reasoning about circular liveness dependencies in model checking. Welder's composition rule Compose-Dep (eq. (6)) focuses on acyclic liveness dependencies. This rule suffices in practice because controllers typically form an acyclic liveness dependency hierarchy. For circular liveness dependencies, developers need to reason about non-waiting actions first and then chain them with their dependent actions.

Control-plane correctness. Recent studies [42, 58] reported controller interactions as a major cause of real-world failures [3–5, 8, 9, 12, 13]. However, existing testing techniques [41, 42, 74] target individual controllers rather than their interactions. Kivi [58] checks controller interference in a specific deployment at the *model* level, not the implementation level. In contrast, Welder verifies controller implementations and interactions against the CORE specification.

9 Discussion

CORE provides a correctness specification that captures controllers' essential functionality and composition requirements, but it does not aim for specification completeness or behavioral conformance. It does not fully determine a controller's behavior; for example, it does not directly constrain the order in which a controller issues different requests. Nor does CORE require verified controllers to exhibit the same behavior as their reference implementations. Instead, CORE's feature coverage is determined by the definition of the `match` predicate in ESR: if a feature is omitted from `match`, CORE does not guarantee that the verified controller implements it correctly. CORE provides flexibility for developers to add features to a verified controller gradually.

Using Welder to verify CORE is not an easy task. It requires expertise in TLA and Verus as required by Anvil. Our experience shows that AI agents with frontier models still cannot completely automate the proof of CORE, but they are good at completing certain types of proofs with our guidance, including (1) finishing the invariant proof given all the inductive invariants, (2) generating a liveness proof skeleton for a new controller given an existing liveness proof, and (3) proving TLA proof rule lemmas given existing lemma examples. We expect that the effort of verifying CORE will drop as LLMs are increasingly better at writing proofs.

In future work, we aim to automate proofs for CORE using AI agents. We plan to investigate how to make the proof more modular, e.g., using separation logics. We envision a future in which controller developers integrate Welder verification into their development workflows, enabling controllers to be continuously verified against their CORE specifications just as they are continuously tested today.

10 Concluding Remarks

Welder shows a way to formally verify the entire cluster control plane beyond individual controllers. It does so using a compositional approach based on an open specification. We have built and verified four Kubernetes controllers using Welder, demonstrating a progressive path to verify a cluster control plane by gradually verifying its controllers.

Acknowledgments

We thank our shepherd, Pedro Fonseca, and the anonymous reviewers. We thank the Verus developers for their help. This work was funded in part by NSF CNS-2145295, NSERC RGPIN-2026-06371, ISF 4136/25, the Maimonides Fund's Future Scientists Center, the Center for New Scientists at the Weizmann Institute of Science, and the Azrieli Foundation.

References

[1] Kubernetes Issue: MultiKey RW Transactions + Indexing in Storage. https://github.com/kubernetes/kubernetes/issues/27548, 2016.

[2] Stricter checking/validation of statefulset pvcs. https://github.com/kubernetes/kubernetes/issues/41153, 2017.

[3] How a Production Outage Was Caused Using Kubernetes Pod Priorities. https://grafana.com/blog/how-a-production-outage-was-caused-using-kubernetes-pod-priorities/, 2019.

[4] Replicaset controller bug: continuously creating pod to tainted nodes. https://github.com/kubernetes/kubernetes/issues/75913, 2019.

[5] Scheduler kept assigning pods with high CPU usage to the same node causing a kernel panic and pod failure loop. https://updates.moonlightwork.com/outage-post-mortem-87370, 2019.

[6] Alloy 6. https://alloytools.org/alloy6.html, 2021.

[7] kube-controller-manager: Garbage Collector is removing Resources with owner reference to Custom Resource. https://github.com/kubernetes/kubernetes/issues/98471, 2021.

[8] Descheduler incorrectly work with HPA (when replicas more than number of nodes). https://github.com/kubernetes-sigs/descheduler/issues/921, 2022.

[9] Knative webhook HPA attempt at scaling up apparently reverted by the operator. https://github.com/knative/operator/issues/1200, 2022.

[10] Failing to Auto Scale Elasticsearch in Kubernetes. https://engineering.zalando.com/posts/2024/06/failing-to-auto-scale-elasticsearch-in-kubernetes.html, 2024.

[11] OpenAI API, ChatGPT and Sora Facing Complete Outage due to Kubernetes Control Plane Issues. https://status.openai.com/incidents/01JMYB483C404VMPCW726E8MET/write-up, 2024.

[12] The Grafana operator does not respect/overwrites HPA settings for a deployment. https://github.com/grafana/grafana-operator/issues/1720, 2024.

[13] StatefulSet volumeClaimTemplates validation allows controller to create PVCs, dueling with out-of-band recreation of PVCs. https://github.com/kubernetes/kubernetes/issues/134357, 2025.

[14] Unseen Catalyst: A Simple Rollout Caused a Kubernetes Outage. https://www.reddit.com/r/RedditEng/comments/1id01im/unseen_catalyst_a_simple_rollout_caused_a/, 2025.

[15] Verus discussion: Dual-mode functions. https://github.com/verus-lang/verus/discussions/1429, 2025.

[16] Argo Rollouts Documentation. https://argo-rollouts.readthedocs.io/en/stable/, 2026.

[17] Elastic Cloud on Kubernetes: Nodes Orchestration. https://www.elastic.co/guide/en/cloud-on-k8s/current/k8s-orchestration.html, 2026.

[18] K8ssandra Documentation: Cass Operator. https://docs.k8ssandra.io/components/cass-operator/, 2026.

[19] kube-rs/kube: Rust Kubernetes client and controller runtime. https://github.com/kube-rs/kube, 2026.

[20] Kubernetes Deployment. https://kubernetes.io/docs/concepts/workloads/controllers/deployment, 2026.

[21] Kubernetes Deployment Controller. https://github.com/kubernetes/kubernetes/tree/v1.35.3/pkg/controller/deployment, 2026.

[22] Kubernetes Parent Reference. https://kubernetes.io/docs/concepts/overview/working-with-objects/owners-dependents/, 2026.

[23] Kubernetes ReplicaSet. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset, 2026.

[24] Kubernetes ReplicaSet Controller. https://github.com/kubernetes/kubernetes/tree/v1.35.3/pkg/controller/replicaset, 2026.

[25] Kubernetes StatefulSet. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset, 2026.

[26] Kubernetes StatefulSet Controller. https://github.com/kubernetes/kubernetes/tree/v1.35.3/pkg/controller/statefulset, 2026.

[27] MongoDB Community Operator Code. https://github.com/mongodb/mongodb-kubernetes/blob/a0ccb1ec0313796a2165d37e490a6878e7a4f16c/mongodb-community-operator/controllers/replica_set_controller.go#L227-L246, 2026.

[28] Possible name collision between RS and STS controllers. https://github.com/kubernetes/kubernetes/issues/138038, 2026.

[29] RabbitMQ. https://www.rabbitmq.com/, 2026.

[30] TiDB Operator Code. https://github.com/pingcap/tidb-operator/blob/d03728387d0a9db8477fae102e79b417ba31cbd1/pkg/controllers/tikvgroup/tasks/updater.go#L89-L108, 2026.

[31] Berner, C. Scaling Kubernetes to 2,500 Nodes. https://openai.com/index/scaling-kubernetes-to-2500-nodes/, Jan. 2018.

[32] Brunel, J., Chemouil, D., Cunha, A., and Macedo, N. The Electrum Analyzer: Model Checking Relational First-Order Temporal Specifications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)* (Sept. 2018).

[33] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. *Communications of the ACM 59*, 5 (May 2016), 50–57.

[34] Chajed, T., Tassarotti, J., Kaashoek, M. F., and Zeldovich, N. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[35] Chajed, T., Tassarotti, J., Theng, M., Jung, R., Kaashoek, M. F., and Zeldovich, N. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).

[36] Chajed, T., Tassarotti, J., Theng, M., Kaashoek, M. F., and Zeldovich, N. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[37] Chang, Y.-S., Jung, R., Sharma, U., Tassarotti, J., Kaashoek, M. F., and Zeldovich, N. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)* (July 2023).

[38] Chen, H., Chajed, T., Konradi, A., Wang, S., Ileri, A., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[39] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[40] Chen, X., Li, Z., Zhang, J., Narayanan, V., and Burtsev, A. Atmosphere: Practical Verified Kernels with Rust and Verus. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP'25)* (Oct. 2025).

[41] Gu, J. T., Sun, X., Zhang, W., Jiang, Y., Wang, C., Vaziri, M., Legunsen, O., and Xu, T. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[42] Gu, J. T., Tang, Z., Su, Y., Stoica, B. A., Sun, X., Zheng, W. X., Zhang, Y., Rahman, A., Wang, C., and Xu, T. Who Watches the Watchers? On the Reliability of Softwarizing Cloud Application Management. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)* (May 2026).

[43] Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., and Parno, B. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[44] Hance, T., Zhou, Y., Lattuada, A., Achermann, R., Conway, A., Stutsman, R., Zellweger, G., Hawblitzel, C., Howell, J., and Parno, B. Sharding the State Machine: Automated Modular Reasoning

for Complex Concurrent Systems. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)* (July 2023).

[45] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[46] Jones, C. B. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 5*, 4 (Oct. 1983), 596–619.

[47] Jung, R., Krebbers, R., Jourdan, J.-H., Bizjak, A., Birkedal, L., and Dreyer, D. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming 28* (2018), e20.

[48] Kim, M., Shin, A., Maeng, J., Jeon, M., and Chun, B.-G. Garen: Reliable Cluster Management with Atomic State Reconciliation. In *Proceedings of the 21st European Conference on Computer Systems (EuroSys'26)* (Apr. 2026).

[49] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)* (Oct. 2009).

[50] Lamport, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923.

[51] Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J. R., Padon, O., and Parno, B. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP'24)* (Nov. 2024).

[52] Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying Rust Programs Using Linear Ghost Types. In *Proceedings of 2023 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'23)* (Apr. 2023).

[53] LeBlanc, H., Lorch, J. R., Hawblitzel, C., Huang, C., Tao, Y., Zeldovich, N., and Chidambaram, V. PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI'25)* (July 2025).

[54] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)* (Apr. 2010).

[55] Leroy, X. Formal Verification of a Realistic Compiler. *Communications of the ACM 52*, 7 (July 2009), 107–115.

[56] Leung, D., Zeldovich, N., and Kaashoek, F. Shipwright: Proving Liveness of Distributed Systems with Byzantine Participants. *arXiv:2507.14080* (July 2025).

[57] Liu, B., Kheradmand, A., Caesar, M., and Godfrey, P. B. Towards Verified Self-Driving Infrastructure. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)* (Nov. 2020).

[58] Liu, B., Lim, G., Beckett, R., and Godfrey, P. B. Kivi: Verification for Cluster Management. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC'24)* (July 2024).

[59] Lorch, J. R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J. R., and Zhao, X. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)* (June 2020).

[60] Macedo, N., Brunel, J., Chemouil, D., and Cunha, A. Pardinus: A Temporal Relational Model Finder. *Journal of Automated Reasoning 66*, 4 (Nov. 2022), 861–904.

[61] Madhu, C. Preventing Controller Sprawl From Taking Down Your Cluster. https://youtu.be/fu5GXo7jmV0?t=732, Oct. 2022.

[62] McMillan, K. L. Circular Compositional Reasoning about Liveness. In *Proceedings of the 10th Conference on Correct Hardware Design and Verification Methods (CHARME'99)* (Sept. 1999).

[63] McMillan, K. L., and Padon, O. Ivy: A Multi-Modal Verification Tool for Distributed Algorithms. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV'20)* (July 2020).

[64] Melissaris, T., Nabar, K., Radut, R., Rehmtulla, S., Shi, A., Chandrashekar, S., and Papapanagiotou, I. Elastic Cloud Services: Scaling Snowflake's Control Plane. In *Proceedings of the 13th ACM Symposium on Cloud Computing (SOCC'22)* (Nov. 2022).

[65] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[66] O'Hearn, P. W. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science 375*, 1-3 (Apr. 2007), 271–307.

[67] Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., and Shoham, S. Reducing Liveness to Safety in First-Order Logic. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[68] Padon, O., Hoenicke, J., McMillan, K. L., Podelski, A., Sagiv, M., and Shoham, S. Temporal Prophecy for Proving Temporal Properties of Infinite-State Systems. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD'18)* (Oct. 2018).

[69] Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)* (June 2016).

[70] Qiu, L., Kim, Y., Shin, J.-Y., Kim, J., Honoré, W., and Shao, Z. LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'24)* (June 2024).

[71] Sharma, U., Jung, R., Tassarotti, J., Kaashoek, F., and Zeldovich, N. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[72] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[73] Spies, S., Gäher, L., Gratzer, D., Tassarotti, J., Krebbers, R., Dreyer, D., and Birkedal, L. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)* (June 2021).

[74] Sun, X., Luo, W., Gu, J. T., Ganesan, A., Alagappan, R., Gasch, M., Suresh, L., and Xu, T. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[75] Sun, X., Ma, W., Gu, J. T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., and Xu, T. Anvil: Verifying Liveness of Cluster Management Controllers. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)* (July 2024).

[76] Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., and Zhang, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov.

2020).

[77] Tang, L., Bhandari, C., Zhang, Y., Karanika, A., Ji, S., Gupta, I., and Xu, T. Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)* (May 2023).

[78] Taube, M., Losa, G., McMillan, K. L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J. R., and Woos, D. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)* (June 2018).

[79] Timany, A., Gregersen, S. O., Stefanesco, L., Hinrichsen, J. K., Gondelman, L., Nieto, A., and Birkedal, L. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. In *Proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'24)* (Jan. 2024).

[80] Vafeiadis, V., and Parkinson, M. A Marriage of Rely/Guarantee and Separation Logic. In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR'07)* (Sept. 2007).

[81] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)* (Apr. 2015).

[82] Wickerson, J., Dodds, M., and Parkinson, M. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)* (Mar. 2010).

[83] Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., and Anderson, T. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)* (June 2015).

[84] Yao, J., Tao, R., Gu, R., and Nieh, J. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. In *Proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'24)* (Jan. 2024).

[85] Zhang, J., Xu, X., Zou, Y., Tang, Z., Wan, X., Hu, K., Wang, S., Xu, W., Wang, D., Chen, H., Huang, L., Yan, S., Tamir, Y., Luo, Y., Wang, X., Yu, H., Wang, Z., Tian, H., and Zhou, D. CortenMM: Efficient Memory Management with Strong Correctness Guarantees. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP'25)* (Oct. 2025).

[86] Zhang, Z., Zhou, T., Jenkins, C., Chowdhury, O., and Mu, S. AutoMan: Facilitating Verified Distributed Systems Development Through Automatic Code Generation and Manual Optimizations. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP'25)* (Oct. 2025).

[87] Zhao, Q., Pîrlea, G., Grzeszkiewicz, K., Gilbert, S., and Sergey, I. Compositional Verification of Composite Byzantine Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS'24)* (Dec. 2024).

[88] Zhou, Z., Anjali, Chen, W., Gong, S., Hawblitzel, C., and Cui, W. VeriSMo: A Verified Security Module for Confidential VMs. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)* (July 2024).

[89] Zou, M., Ding, H., Du, D., Fu, M., Gu, R., and Chen, H. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[90] Zou, M., Du, D., Dong, M., and Chen, H. Using Dynamically Layered Definite Releases for Verifying the RefFS File System. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)* (July 2024).