Research Statement

Xudong Sun (UIUC)

I am a *computer systems* researcher with a focus on system reliability. I develop *formal verification* and *software testing* techniques to address pressing reliability issues in cloud systems.

The reliability of cloud systems is paramount. Modern clouds are architected as layers of critical systems including cluster managers, storage systems, databases, big data and streaming execution engines, service meshes, machine learning systems, etc. Defects in these systems can cause disastrous failures such as service outages, data loss, and security issues. However, ensuring the reliability of cloud systems is notoriously hard. First, many of these systems do not have a clearly defined correctness specification, making it hard to test these systems rigorously, let alone formally verify them. Second, cloud systems run within complex and dynamic environments: They need to handle unusual user input, and tolerate various factors including unexpected failures, network issues, concurrency, and asynchrony. As a result, testing often falls short due to the low coverage, and formal verification is hard to apply to practical systems.

My research goal is to improve the reliability of modern clouds, and my vision is to formally verify the entire cloud stack. To be practical, my research leverages an opportunity that modern cloud systems are increasingly architected as a fleet of modularized components instead of being monolithic, so that verification can be made practical by focusing on verifying each component at a time and then composing their proofs to obtain a correctness guarantee of the entire system. In this way, we can gradually replace existing components with formally verified ones. For example, my previous work Anvil is a framework that allows developers to implement cluster manager components in Rust and verify that their implementations satisfy high-level properties including liveness and safety [10]. The verified components can be directly deployed in a real-world Kubernetes cluster and used as drop-in replacements for existing, unverified components; the verified components achieve feature parity and the same performance compared to existing ones. Anvil's verification guarantees the absence of a broad range of bugs, including most bugs detected by my work on testing cluster managers [3, 8, 11]. In addition to cluster managers, my research covered other layers and aspects of modern cloud stack, including consensus-based storage systems (e.g., ZooKeeper) [4,5], cloud service (e.g., Azure Storage, AWS S3) integrations [1], and cloud system configurations [6]. My research has involved techniques including deductive verification [10], model checking [4, 5], fault injection [1, 8, 11], and software testing [3, 6].

My research methodology is to first develop a deep understanding of real-world, widely-used cloud systems by systematic testing, and then build provably correct, practical cloud systems by formal verification. My insight is that the key to verifying practical systems is not only about writing proofs, but also about coming up with formal specifications. In practice, many important systems do not have a formal specification; they are not built by referring to well-studied algorithms or protocols from textbooks or research papers—they are built with developers' best-effort heuristics and documented in ambiguous prose. I believe that testing is a practical approach to understanding what key properties a system should guarantee but often fails to guarantee. For example, my previous research, Sieve [8] and Acto [3], tested over 20 different components in Kubernetes and detected over 120 severe new bugs. The testing results revealed a dominating failure pattern: Cluster managers fail to eventually achieve the desired state, which is a liveness violation. This insight motivates the Anvil project and leads me to formalize the correctness of cluster managers as Eventually Stable Reconciliation (ESR), a liveness property. ESR has already turned into practical systems—I have built three microservices running as part of Kubernetes and verified that they implement ESR using Anvil.

My research emphasizes making real-world impact in both academia and industry. Through automatic testing and model checking, my research has discovered over 200 new bugs and led to over 120 bug fixes in critical cloud systems, including Kubernetes, ZooKeeper, Hadoop, HBase, Alluxio, Orleans, and cluster manager microservices for managing popular cloud systems, including Cassandra, Elasticsearch, Knative, MongoDB, RabbitMQ, Redis, TiDB, and ZooKeeper. Many of the bugs have severe consequences, including application outages, data loss, and security issues. Some of the bugs reflect design-level flaws: To fix these bugs, developers revamped the original design by changing hundreds of lines of implementation. Developers highly appreciated our bug reports with comments like "It is a great analysis and verification!" The ideas of Ctest, Sieve, and Acto have been adopted and followed by developers from large companies including Anyscale, Google, and VMware. We have been invited to present Acto, Sieve, and Anvil at major industrial conferences including KubeCon and SRECon, and publish articles at the USENIX ;login: magazine [2, 7, 9]. Our outreach has sparked interest from open-source communities (e.g., Elasticsearch) and companies (e.g., Amazon) in adopting our research. Ctest, Sieve, and Acto are used to develop course projects for two courses on software testing and operating systems at UIUC, respectively. Anvil received the Jay Lepreau Best Paper Award at OSDI 2024. All my research projects are open-sourced to enable further research and development.

1 Past and Current Research

Testing cluster management controllers. Modern cluster managers break down management logic into a fleet of microservices, called controllers. For example, in Kubernetes, all the cluster management logic is encoded in different controllers. Today, thousands of controllers are implemented by commercial vendors and open-source communities to extend Kubernetes with new capabilities. Despite the importance and prevalence of controllers, ensuring their reliability is challenging. Controllers run within complex, dynamic, and distributed environments. They must safely drive the system to desired states while tolerating unexpected failures, network interruptions, and asynchrony issues. Unreliable controllers lead to severe consequences such as application outages, data loss, and security issues.

I observed that controllers' reliability challenges come from their *state-reconciliation* design: Each controller continuously monitors a subset of the cluster state and reconciles the *current* cluster state to match a *desired* state. A controller's behavior depends on the cluster state it observes, but there is an enormous space of cluster states that a controller can possibly observe, making it hard to design a state-reconciliation procedure that never goes wrong. Based on my observation, I built **Sieve, the first automatic reliability testing technique for cluster management controllers** [8]. The key idea of Sieve is to drive unmodified controllers to their buggy corners by systematically perturbing the controller's view of the cluster state. Sieve performs *bounded exhaustive* testing and does not depend on hypotheses about vulnerable regions in the code where bugs may lie. To detect diverse controller bugs, Sieve employs three perturbation patterns that expose controllers to (1) intermediate states, i.e., cluster states resulted from the controller crashing in the middle of state reconciliation, (2) stale states (or past cluster states) due to asynchrony and slowness issues, and (3) unobserved states caused by missing some cluster state transitions due to data race or network issues. To automatically flag a triggered bug, Sieve uses differential oracles that compare the cluster state's evolution with and without perturbations to automatically detect safety and liveness issues. To help developers localize each bug in the code, Sieve can deterministically reproduce previously triggered bugs by replaying the corresponding state perturbations. Sieve has discovered 46 serious new bugs (35 confirmed and 22 fixed) in 10 popular Kubernetes controllers for managing critical cloud systems including Cassandra, MongoDB, and ZooKeeper.

Sieve is highly automated but its testing quality depends on developer-provided testing workloads. My research then introduced **Acto, the first functional testing tool that automatically generates high-coverage testing workloads for controllers** [3]. Acto generates hundreds of diverse and representative workloads to test whether a controller can reconcile the cluster state to match different desired states, and whether a controller can reach the same desired state starting from different initial cluster states. Acto has discovered 81 serious new bugs (62 confirmed and 43 fixed) in 14 popular Kubernetes controllers.

Building formally verified, practical cluster management controllers. Testing controllers revealed a dominating failure pattern: Controllers fail to eventually achieve the desired state, which is a liveness violation. With lessons learned from testing, I built **Anvil, the first framework for building formally verified, practical cluster management controller implementations** [10]. Anvil allows developers to confirm controller correctness at compile time using formal verification: Developers write the controller implementation along with a proof that the implementation always meets a specification (e.g., liveness or safety) of the controller's desired behavior. The verified controllers are practical and can be directly deployed in a real-world Kubernetes cluster and manage real-world applications.

To enable formal verification for controllers and reduce developers' specification burden, Anvil formalized controller correctness into *Eventually Stable Reconciliation (ESR)*, a liveness property written in the Temporal Logic of Actions (TLA). ESR states that a controller should *eventually* reconcile the cluster to a desired state, and then *always* keep the cluster in the desired state. ESR is generally applicable to all controllers and it captures the essential functionality that controllers should provide in a precise language. ESR precludes a broad range of bugs caused by factors like inopportune failures, network disruptions, concurrency issues, and conflicts with other controllers.

To reduce developers' proof burden, Anvil provides a set of features. A unique challenge of proving liveness properties such as ESR is that liveness depends on subtle *fairness assumptions*, including assumptions about possible failures (e.g., crashes due to hardware or power failures). Overly strong assumptions (e.g., the controller can crash at most once) lead to weak correctness guarantees, and overly weak assumptions (e.g., the controller can keep crashing forever) make liveness verification impossible—when running on a machine that keeps crashing immediately after reboot, even a correct controller cannot make progress to the desired state. Anvil employs an assumption that covers a broad range of failure scenarios—an arbitrary number of failures can happen, but eventually failures stop happening. In addition, Anvil provides a temporal embedding and a set of temporal lemmas to facilitate and automate liveness proofs for controllers, a general proof strategy to help developers structure their ESR proofs, and a general environment model that encodes the (trusted) system components that interact with controllers and the failure model of the controllers.

With Anvil, I have built three practical Kubernetes controllers for managing ZooKeeper, RabbitMQ, and FluentBit and verified that they implement the ESR property. Although Anvil is primarily designed for liveness verification, it also supports *safety* verification; I proved a safety property specific to the RabbitMQ controller. The verified controllers can readily be deployed in real-world Kubernetes platforms and provide feature parity w.r.t. existing mature, widely used (but unverified) controllers; they implement critical features including scaling, upgrading, and reconfiguration. The verification process also exposed deep bugs in both our early implementations and unverified reference controllers. Evaluation shows that that the verified controllers achieve the same performance compared to the widely-used, unverified references. Testing the verified controllers using Sieve and Acto finds no bugs in the verified code.

I have been continuously developing and improving Anvil to make it an open platform for developing cluster management controllers and verifying various liveness and safety properties. I am using Anvil to implement and verify a series of core controllers in Kubernetes (e.g., ReplicaSet, StatefulSet). These controllers directly manage low-level cluster resources including nodes and containers, and are used to support the higher-level controllers that manage applications, e.g., the aforementioned three controllers. In addition, I am using Anvil to explore exciting future research directions (§2). I envision in the future Anvil being the default platform that practitioners use to build their controllers, and researchers use to experiment with different verification techniques (e.g., automating liveness proof).

Efficient model checking for consensus-based storage systems. Consensus-based storage systems, such as etcd and ZooKeeper, play a critical role in today's cloud stack. For example, they are used to maintain and serve the cluster state to cluster managers like Kubernetes. These systems are notoriously hard to get right because they implement complex consensus algorithms, fault tolerance mechanisms, and performance optimizations. To efficiently verify these systems' implementations, my research has introduced efficient model checking techniques [4, 5]. For example, Remix [4] allows developers to verify consensus-based system implementations using specification-level model checking and conformance testing. With Remix's methodology, developers write multi-grained specifications, i.e., multiple specifications with different granularities for composable modules, and compose them into mixed-grained specifications for specific scenarios. For example, to verify a code change, developers compose a mixed-grained specification using fine-grained specifications of changed modules and coarse-grained specifications that abstract away details of unchanged code. Remix performs conformance testing to ensure that each specification conforms to the corresponding implementation, performs model checking using the mixed-grained specification, and confirms any bug found during specification-level model checking by replaying it at the implementation level. The model checking techniques have discovered 29 bugs in widely-used consensus-based storage systems that implement protocols such as Raft and Zab.

Testing cloud-backed applications. Modern applications increasingly depend on cloud services (e.g., AWS S3, Azure Storage) for various functionalities. Despite the benefits, such "cloud native" practice imposes emerging reliability challenges introduced by the fault models of opaque cloud backends and less predictable connections between the application and cloud services. My research takes a first step to address the emerging reliability challenges by building a "push-button" reliability testing tool named Rainmaker [1], as a basic SDK utility for any cloud-backed application. Rainmaker helps developers easily and systematically test their applications' correctness, in the face of various errors under the cloud-based fault model. The core of Rainmaker is its automatic fault injection policies that define what faults to inject and where (e.g., at which REST calls) to inject faults. Rainmaker's fault injection policies are guided by a bug taxonomy: It considers transient error(s) that can occur during one REST API call initiated by the application (and the corresponding retries by the SDK); yet, it captures common bug patterns and shows that error (mis)handling of even one REST call can have major impacts on application correctness. Rainmaker has detected 73 bugs (55 confirmed and 51 fixed) in 11 popular cloud-backed applications.

Testing configuration changes in cloud systems. Besides code bugs, configuration errors are among the dominant causes of cloud and datacenter failures, and they are notoriously fatal and hard to deal with. The problem is exacerbated by the high speed of configuration changes—large-scale cloud services deploy hundreds to thousands of configuration changes to production systems daily, often more frequent than code changes. My research introduced Ctest [6], a new type of tests for detecting failure-inducing configuration changes. Ctests connect software tests with production system configurations and test configuration changes in the context of code that is affected by the changes. In this way, Ctests effectively detect both configuration errors and code bugs triggered by configuration changes. My research also introduced a methodology for generating Ctests by transforming the existing and abundant tests in mature software projects in an automated fashion that reuses well-engineered test logic and oracles. I evaluated Ctests using 64 real-world configuration-induced failures, 1,055 diverse misconfigurations generated by error injection rules, and 92 deployed configuration files from publicly-available Docker images. Ctests detected 96.9% of the real-world failure-inducing configurations, 72% of the generated misconfigurations, and 10 misconfigurations in 7 deployed files.

2 Future Research

My overarching goal is to build reliable cloud systems. I will continue my efforts toward this goal and expand my research to reliability challenges in new domains. Here is an outline of my future research directions.

Compositional verification for cloud systems. Cloud systems are increasingly architected as a fleet of modularized components. Verifying the system requires compositional verification of the interacting components. For example, cluster managers are composed of layers of controllers. My previous work [10] allows developers to formally verify cluster management controllers separately, but that does not imply the correctness of the entire cluster manager. I plan to build compositional verification techniques to reason about controllers and their interactions collectively. This verification is challenging for mainly two reasons: (1) Controller correctness is formalized as liveness, and composing liveness is hard due to circularity, and (2) controller interactions are asynchronous—instead of directly calling other controllers, a controller interacts with other controllers by monitoring changes to the cluster state made by other controllers and issues new changes to the cluster state monitored by other controllers. To address these challenges, I will (1) design composable liveness specifications and proof techniques to reason about controller liveness dependencies, and (2) formalize a notion of ownership—a subset of the cluster state that is exclusively managed by some controller, and use this ownership to prove that controllers do not interfere with each other.

Pushing systems verification into practice by reducing proof burden. My previous work Anvil [10] enables formal verification for practical cluster management controller implementations, but it has a scalability issue because developers need to spend extensive manual proof effort—a common problem in deductive verification. I plan to work on reducing developers' proof burden by automating proofs, especially liveness proof for complex system implementations; most of the previous work focused on automating safety proof, but not liveness. As a concrete first step, I will work on automating liveness verification for controllers in Anvil. From my experience, liveness verification for system implementations is especially challenging: Proving liveness requires temporal logic reasoning which is not natively supported by existing SMT-based automated theorem provers (e.g., Z3). To achieve better automation, I plan to offload temporal logic reasoning to dedicated temporal verifiers. A key challenge is to connect existing proofs in Verus to temporal verifiers while ensuring soundness. I also plan to explore other options, such as using ranking functions to reduce the liveness property into safety properties that are easier to verify, fitting the proof into a decidable fragment of first-order logic (e.g., EPR), and using LLM-based approaches for proof generation, synthesis, and repair.

Enabling verification for real-world, complex systems by developing specifications. I am excited to work on developing formal specifications for critical cloud systems. Many real-world, complex systems do not have formal specifications of their correctness. Lacking specifications disables formal verification and also makes testing challenging—it is hard to write comprehensive testing oracles without a specification. For example, I plan to develop specifications for commonly used cluster schedulers (e.g., Kubernetes schedulers). These schedulers should place application tasks on cluster nodes while satisfying a series of constraints, including affinity and load balance—a challenging constraint satisfaction problem. Designing specifications for systems like cluster schedulers brings many interesting research problems, such as deciding the specification style, disambiguating system design choices that are poorly documented, and validating the specifications. The specifications can be used in many ways, such as top-level theorems for verification, oracles for testing, and environment models and mocks for verifying and testing upper-layer applications.

System reliability under emerging computing paradigms. The way people develop and deploy software systems is shifting rapidly with emerging computing paradigms. Under the "cloud native" computing paradigm, applications are often split into fine-grained components (e.g., microservices, serverless functions), depend on cloud services (e.g., AWS S3, Azure Storage) for various functionalities, and are deployed and managed using cluster managers (e.g., Kubernetes). With recent trends of Hybrid Cloud and Sky Computing, an application might depend on services spanning across different clouds. The new computing paradigms bring many benefits including availability, scalability, and flexibility, but also make reliability more challenging than before. Under the new paradigms, applications' failure models are complicated by the many independent and interacting system components and the opaque cloud backends, and new failure patterns such as cross-system interaction failures are increasingly prevalent. For example, when relying on cloud services that can fail independently, an application needs to correctly interpret and handle different errors from the underlying services to tolerate transient failures and avoid idempotence and consistency issues. Applications in new domains are also experiencing a paradigm shift. For example, modern AI systems are increasingly built as compound systems with multiple interacting components [12]. **A long-term goal of my research is to study and address the new reliability issues brought by the emerging computing paradigms.** In particular, I plan to work on improving the reliability of applications in new domains, such as AI systems and immersive computing systems.

References

[1] Yinfang Chen, **Xudong Sun**, Suman Nath, Ze Yang, and Tianyin Xu. Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, April 2023.

[2] Jiawei Tyler Gu, **Xudong Sun**, Zhen Tang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. Acto: Push-Button End-to-End Testing for Operation Correctness of Kubernetes Operators. In *USENIX ;login:*, August 2024.

[3] Jiawei Tyler Gu, **Xudong Sun**, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 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)*, October 2023.

[4] Lingzhi Ouyang, **Xudong Sun**, Ruize Tang, Yu Huang, Madhav Jivrajani, Xiaoxing Ma, and Tianyin Xu. Multi-Grained Specifications for Distributed System Model Checking and Verification. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys'25)*, March 2025.

[5] Ruize Tang, **Xudong Sun**, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)*, April 2024.

[6] **Xudong Sun**, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, November 2020.

[7] **Xudong Sun**, Jiawei Tyler Gu, Cody Rivera, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Building Kubernetes Controllers That Do Not Break. In *USENIX ;login:*, June 2024.

[8] **Xudong Sun**, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, July 2022.

[9] **Xudong Sun**, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Sieve: Chaos Testing for Kubernetes Controllers. In *USENIX ;login:*, November 2024.

[10] **Xudong Sun**, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying Liveness of Cluster Management Controllers. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, July 2024.

[11] **Xudong Sun**, Lalith Suresh, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lilia Tang, and Tianyin Xu. Reasoning about Modern Datacenter Infrastructures Using Partial Histories. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)*, June 2021.

[12] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.