Lessons Learned from Incorporating Formal Methods in Huawei Cloud Reliability

Claudia Cauli[*1] Timo Lang[*1] Shuo Chen[2]

Sebti Mouelhi[1] Xin Jin[2] Subhajit Bandopadhyay[1]

Xusheng Chen[2] Yazhi Feng[2] Haoze Song[2]

Linhua Tang[1] Zhenli Sheng[2‡] Ananth Shrinivas Srinath[1‡]

Abstract

Formal methods are increasingly adopted in systems where reliability and correctness are critical, enabled by improvements in tool usability, speed, and automation. This industrial experience report presents three projects at Huawei Cloud showcasing different trade-offs in investment and assurance levels. We applied probabilistic concurrency testing, model checking, and deductive verification to two foundational services in the database and networking domains: the K2 transactional key-value store and the Global Server Load Balancer (GSLB).

Our work discovered bugs across different system layers—from design to implementation—which led to enhanced reliability, protocol simplifications, improved understanding, and generally higher developer confidence.

To support practical adoption, we provide empirical and anecdotal data—including person-month effort metrics, tool comparisons, and success patterns—enabling engineering teams to make informed decisions about which approaches to invest in, when to apply lightweight versus heavyweight techniques, and how to budget for formal methods adoption in their organisation.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Software and its engineering → Formal methods**; • **Theory of computation → Automated reasoning**; **Logic and verification**.

Keywords: Industrial Experience, Formal Methods, Cloud Reliability

[*]Corresponding authors: {claudia.cauli, timo.lang}@huawei.com
[1]Huawei Technologies Co., Ltd, Dublin, Ireland
[2]Huawei Technologies Co., Ltd, Shenzhen, China
[‡]Equal senior contributions

This work is licensed under a Creative Commons Attribution 4.0 International License.

EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3803626

1 Introduction

In software and hardware engineering, the term *formal methods* (FM) refers to techniques that rely on exact mathematical representations and logical reasoning as their foundation to establish system correctness. Correctness varies depending on the system being analysed and is formulated through precise specifications that are automatically checked against some formal representation. Unlike traditional testing, formal methods systematically or exhaustively explore the system's state space, allowing practitioners to gain higher confidence in its correctness. Some approaches prioritise usability and accessibility over analysis completeness (so-called *lightweight* formal methods) while others prioritise analysis completeness above all else (so-called *heavyweight* formal methods) [1].

Starting in the 90s, early formal methods adoption was largely confined to safety-critical domains, like aerospace and defence, where the complexity of formal verification could be justified by catastrophic failure risks. Throughout the past thirty years, advances in usability, automated tooling, scalable model checkers, and lightweight techniques, have dramatically lowered the barrier to entry—enabling their adoption in fast-paced software and Cloud industries [2]. Companies like Microsoft, Amazon, and Google now routinely apply formal methods to distributed systems, protocols, configurations, and cloud infrastructure, for both internal developer tools and external customer-facing products [3–5]. The complexity of these systems, paired with strict data management and security compliance regulations, creates a compelling case for mathematical rigor even outside the traditional safety-critical domains.

Huawei Cloud operates large-scale distributed infrastructure that faces correctness challenges arising from concurrency, non-determinism, and partial failures in distributed systems. These fundamental challenges manifest as reliability issues through customer-impacting bugs and service outages that are often difficult to diagnose, fix and verify using conventional approaches. To address these limitations, the organisation established a formal methods team in 2024 to target critical infrastructure components where correctness bugs carry high operational or reputational costs. The

team adopted a strategic approach, selecting projects based on design complexity, high availability requirements, and the need for algorithmic correctness guarantees. This paper presents three projects that demonstrate practical formal verification value within three cloud infrastructure settings:

1. the transaction and replication protocols for the K2 distributed database (modelled with P and TLA$^+$);
2. the gossip protocol for the peer-to-peer datastore deployed in the GSLB service (modelled with P); and
3. the task sharding algorithm within GLSB (verified with Gobra).

In these projects, the team was able to identify bugs and reliability issues that were later fixed by the developers.

We present a comprehensive analysis of the projects, including workflows, outcomes, challenges, and quantifiable metrics like time, personnel, and engineer feedback. With this report, we aim to provide practical guidance for developers seeking to adopt formal methods efficiently, and to instill curiosity amongst those less familiar with the field.

Structure of this article. Section 2 introduces the systems and tools used for our formal methods effort. It contains no original work. Sections 3-5 describe the three projects in detail, following a common structure. We start by outlining the project's motivation and goals. We follow with a detailed account of the verification process, emphasising modelling choices and documenting challenges. We conclude with key findings and data on team composition and resource allocation. Section 6 synthesises observations common across all projects, distills lessons learned, and provides practical guidance for similar formal methods initiatives. It includes tabular summaries of the issues discovered and project management metrics. Section 7 offers some concluding remarks.

2 Systems and Tools Description

This section sets the scene by introducing the cloud systems and the formal methods tools that are central to the experience described in the paper.

2.1 Systems: K2 and GSLB

The three projects presented in this paper targeted two large-scale systems, the novel distributed database K2 and the Global Server Load Balancer GSLB, and three of their subcomponents: the two core protocols of K2, the core protocol of the P2P datastore in GSLB, and a task sharding algorithm, also part of the GSLB service.

2.1.1 K2. The global transactional key-value store K2 is currently under development at Huawei Cloud [6]. Positioned as the 'Data Root Service' of Huawei Cloud, K2 aims to provide a foundational data management platform comparable to state-of-the-art systems such as Google Spanner [7]. Its

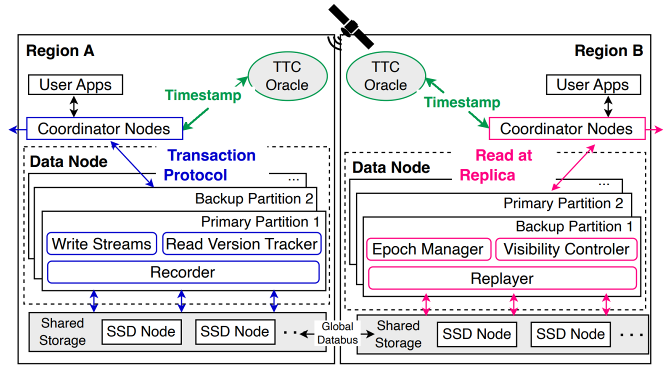

Figure 1. K2 system architecture (adapted from [6]). Region A hosts the primary replica running the K2-MVTO transaction protocol. Region B hosts a secondary replica managed by the K2-VCR replication protocol for consistent reads.

design emphasises global scalability, strong transactional semantics, and high availability, making it suitable for mission-critical enterprise workloads. The architecture of K2 is based on geographically distributed replication. A primary replica is placed in proximity to the customer in order to minimise request latency, while secondary replicas are distributed across nearby regions to ensure efficient access and fault tolerance (see Fig. 1). This topology enables K2 to simultaneously achieve low-latency performance and high throughput, a property essential for modern real-time global applications. At the core of K2 are two protocols: (1) the multi-version timestamp ordering protocol K2-MVTO, which manages the lifecycle of distributed transactions across multiple nodes of the database, and (2) the visibility control protocol K2-VCR, that determines how reads are served to replica nodes.

K2 is designed to achieve five nines (99.999%) availability. The system incorporates mechanisms for rapid fault detection and recovery, enabling resilience against both single-node and availability zone (AZ) failures.

The K2-MVTO Distributed Transaction Protocol. Transactions consist of multiple read and write operations that must be applied to the database atomically (all or none). Read operations specify the keys to be accessed. Write operations also define the values to be written to those keys. During distributed execution, the overall status of a transaction is maintained by a single designated node, referred to as the *Transaction Record Holder* (TRH) or coordinator. Different nodes handle different shards of keys, and when some operation within a transaction attempts to access a given key, that operation will be forwarded to the responsible node, which can be generally called a *participant* node. The lifecycle of a full transaction is as follows:

1. `beginTxn`: the transaction starts and is assigned a TRH;

2. `read/write`: a sequence of operations are issued to the nodes, with each being redirected to its responsible participant node;
3. `commitTxn`: a decision is made to either *commit* or *abort* the transaction, and the states of both the transaction and its write operations are *updated* accordingly; and
4. `finalizeTxn`: the commit or abort decision is then *finalized* asynchronously by either applying the changes to the database or discarding them.

While a single, non-overlapping, transaction is straightforward, concurrent transactions across multiple nodes introduce conflicts that must be resolved to maintain system-level guarantees. Two primary conflict types arise when transactions access the same keys: *read–write* conflicts, where a transaction modifies a key being read by another, and *write–write* conflicts, where a key is modified simultaneously by multiple transactions. Conflicts are resolved using transaction *priority* and *timestamp*: higher-priority transactions take precedence, and equal-priority transactions are resolved in favor of the earlier timestamp. Timestamps are strictly monotonically increasing.

The K2-VCR Replication Protocol. Like similar multi-region data stores, K2 deploys cross-region replicas asynchronously to serve near-client reads. In K2, this replication is managed by the K2-VCR protocol, a new visibility control protocol that makes critical use of true-time clocks. K2-VCR uses epochs (time intervals of fixed length) to index data versions. Epochs are cut independently on each node making use of true-time clocks. The epoch assigned to a transaction is determined collaboratively between all data nodes involved in the transaction. All events that happened on a primary node are logged and transmitted over the network, in chunks reflecting epoch cuts, to be replayed by the replica node. Before serving replica reads, it is enforced that the replica node has replayed all relevant epochs.

2.1.2 GSLB.

Global Server Load Balancing (GSLB) is the practice of distributing and balancing network traffic across multiple geographically distributed data centers and servers. GSLB services aim to improve availability, reduce latency, and ensure performance. Their reliability is utmost, as disruptions may cause inconsistent service behaviour, significant outages, and customer impact. A GSLB service for Huawei Cloud was developed five years ago.

The P2P datastore. A critical component of the GSLB control plane is the decentralized peer-to-peer (P2P) datastore, ensuring eventual synchronisation of states across nodes through a gossip-based protocol. Each peer stores triplets of the form (*key*, *value*, *version*). Triplet updates propagate across the network until all peers converge, with versioning ensuring consistency by discarding stale information. The datastore handles updates at a rate of ∼200 triplets per second, and each peer maintains the latest versioned value for each key. State is distributed and subjected to strict versioning control, so that stale updates are safely ignored. The system eventually converges.

Communication between peers is organised through *bonds*, bidirectional links where each peer simultaneously acts as client and server. When bonds are reliable, they give rise to a single fully connected *island* encompassing all peers. Under degraded conditions, weak or unstable bonds may temporarily fragment the system into multiple islands, reducing reliability. To cope with such situations, peers periodically run *island discovery*, triggered by *ticks*, which are local, clock-driven events (typically every 1–10s) during which a peer executes maintenance actions, such as attempting new bonds. In an island discovery round, a peer semi-randomly sends a discovery request to its strongly bound peers, which then propagate the request transitively until all reachable peers are covered. Depending on the configuration, this mechanism can estimate only the island size or reconstruct the full island topology, helping optimise data paths within an island and identify bridges to reconnect separate islands.

The Task Sharder Library. Multiple components within the GSLB service depend on the Task Sharder library, which handles the splitting (aka 'sharding') of tasks for these to be handled by different machines. At the time of writing, the task sharder has been running continuously for about three years, with its configuration parameters changing multiple times a day. A configuration of the sharder consists of a number of workers and a set of tasks, each consisting of a number of subtasks. Moreover, for every task two parameters are specified: its *span*, describing over how many epochs one run of its execution should be stretched out, and its *overlap*, denoting how many duplicates of the tasks should be run during that time. The core functionality of the sharder is implemented in function *pickSubtasks* that takes as input a worker and an epoch number, and returns the list of all subtasks that should be executed by that worker in the given epoch. As *pickSubtasks* is fully deterministic, it can in fact be implemented and executed by each worker individually, eliminating the need for a central scheduling unit.

2.2 Tools: P, TLA$^+$, and Gobra

The tools we used are P, TLA$^+$ and Gobra. Both P and TLA$^+$ are considered lightweight methods. Users write models of their design or code, alongside the desired specifications, and then run automated checks to determine whether the models comply with the specs. Gobra is a deductive verifier that checks for correctness directly at the code level, and can be considered as a heavyweight method.

2.2.1 The P Framework.

The *P Framework* [8] is a state-machine-based toolset designed for modelling and verifying distributed systems. P supports *incremental abstraction*, allowing developers to construct stepwise models of real-world systems that gradually capture essential behaviours

while keeping the state space tractable. Its analysis capabilities include *Probabilistic Concurrency Testing* (PCT), which uses probabilistic task scheduling and search prioritisation heuristics to systematically explore state-space areas most likely to contain concurrency-related bugs. Such systematic exploration is not exhaustive: by sampling the state space rather than enumerating it, PCT avoids the state-explosion problem faced by exhaustive model checkers like TLC (see Subsection 2.2.2 below), effectively trading completeness for scalability. The theoretical basis of PCT rests on the concept of bug depth, the minimum number of scheduling constraints needed to expose a bug. A program with n threads and k steps will reveal a bug of depth d with probability at least $1/nk^{d-1}$ [9]. Since many real-world concurrency bugs (e.g., atomicity violations, ordering errors, and deadlocks) have low depth, PCT can detect them efficiently. P has been successfully used in multiple production-scale systems, including Amazon's S3 [10], and more recently DeepSeek's 3FS distributed file system [11].

2.2.2 TLA+. The *Temporal Logic of Actions* TLA⁺ [12] is a formal specification language for distributed system modelling. In TLA⁺, both system models and their desired properties are expressed in the same language. The syntax of TLA⁺ is closer to pure mathematics than to programming. Users preferring a more programming-like syntax can use the alternative language *PlusCal* which automatically transpiles to TLA⁺. The TLA⁺ toolkit includes the bounded model checker TLC that systematically verifies user-defined properties by exhaustively exploring the state space up to specified bounds, employing Breadth-First Search (BFS) as its default strategy. This exhaustive search guarantees that all possible executions within the given bound will be examined. Consequently, TLA⁺ can provide stronger verification guarantees compared to P. An additional practical benefit of BFS, combined with bounded model checking, is that bugs found by TLC will be of minimal depth, and therefore easier to analyse and root-cause. Such exhaustive exploration, albeit bounded, comes at the cost of being susceptible to state-space explosion—when applied to models containing numerous independent variables, the check rapidly becomes computationally intractable for deep execution traces.

TLA⁺ has been successfully applied in industry to verify critical distributed systems and uncover subtle design flaws before production, including: at Amazon Web Services for S3, DynamoDB, and EBS [3]; at Microsoft for Azure CosmosDB [13] and to support an incident-report following a 28-day service outage [14]; at MongoDB for replication protocols [15]; and at Alibaba for the distributed file system within their PolarDB database [16].

2.2.3 Gobra. Gobra [17] is a verification-oriented extension of the Go programming language, recently developed at ETH Zurich. In its core, it relies on the Viper intermediate verification language [18], which translates verification conditions into satisfiability formulas. Gobra supports most built-in Go constructs, in particular Go's local concurrency and memory model. Deductive verifiers like Gobra reason about programs using *preconditions* and *postconditions*, an approach going back to ideas developed in the 1960s by Robert W. Floyd and Tony Hoare [19, 20]. Both pre- and postconditions are annotations for programs that are written in a formal language. The precondition specifies what can be assumed at the call site of the function, and the postcondition then specifies what holds after the program execution. A deductive verifier tries to prove that such a specification is true for a program using logical (symbolic) reasoning. In order to succeed, the verifier sometimes needs further annotations within the program, meaning that code blocks (especially loops) are amended with their own pre- and postconditions.

Of the tools presented here, Gobra gives the strongest guarantees, namely *complete* coverage of all possible states. However, this is only achievable through the arduous process of adding annotations to the code line-by-line, and the annotations are often hard to get right. Notably, Gobra was used to fully verify an Internet router part of the SCION architecture, which uses cryptographic protocols for secure packet forwarding [21].

3 Formal Methods for K2's Core Protocols

This section outlines the formal modelling and verification of the two K2 core protocols, K2-MVTO and K2-VCR. For this work, we used P and TLA⁺ as complementary tools to build both a low-level detailed model and two distinct high-level abstract models, respectively. As a result of this effort, we detected four critical atomicity bugs, which were fixed by developers and prevented from reaching production, and gained clarity on one minor consistency aspect, which sparked conversations with the development team.

Context and Motivation. The correctness of the distributed transaction protocol, K2-MVTO, and replication protocol, K2-VCR, is essential to guarantee the externally-observable behaviour expected by customers, including transaction atomicity, fault tolerance, and low latency. In particular, the team was concerned about two properties: atomicity (ensuring all-or-nothing transaction semantics) and distributed consistency (ensuring reads reflect the latest committed write across replicas). Not ensuring atomicity means that customers might lose their data. Not ensuring consistency means that customers might read stale data. Both properties must hold under all execution scenarios. When this project was agreed, the development team was actively developing K2, had already implemented state-of-the-art unit and integration tests, and considered advanced black-box testing frameworks like Jepsen [22]. However, the team was aware that conventional testing would not cover the entire state space of the protocols, and suffer from human bias and

blind spots, and decided to invest in established formal methods techniques, such as TLA$^+$, to exhaustively explore the protocol's design logic.

Verification Goal and Properties. The goal of this project was to gain confidence in the correctness of K2-MVTO and K2-VCR protocols before pushing the database to production. The core properties that we aimed at verifying are

Atomicity: *Either all or none operations of a transaction are applied to the database.* (1)

Strong Consistency: *After a successful write, any subsequent read accesses the latest version.*[1] (2)

Replica Read Monotonicity: *If transaction T1 precedes transaction T2 on primaries, then replicas reading writes from T2 must also read writes from T1.* (3)

Project Onboarding. K2 is developed in C++20; follows an asynchronous programming paradigm through coroutines and advanced language features; and achieves high-performing local concurrency through the powerful Seastar framework [25]. The codebase spans across 1 627 files and ~318k lines of C++ code, including ~40k lines of comments. To onboard on the project, the K2 team provided us with comprehensive documentation on the system's architecture and the two protocols, in the form of design docs, internal Wiki pages, and slide decks, which was shared asynchronously and delivered during in-person meetings. The K2 team appointed three points of contact, and remained continuously available through weekly syncs and offline messaging.

As this work was conducted in three parts, each carried out by different team members, this section includes individual reporting in Subsections 3.1, 3.2, and 3.3.

3.1 Modelling and Verification: P for K2

This subsection details our approach to formally verifying the K2-MVTO transaction protocol and K2-VCR consistent replication protocol for the K2 distributed database using P. Our methodology prioritises a clear, verifiable and detailed model that remains closely aligned with the system's source code, while abstracting away non-essential complexities, which is designed to verify the correctness of the core design and implementation of K2.

Bridging the Gap: Model-to-Code Conformance. A primary challenge in formal verification is maintaining a meaningful connection between the abstract model and the concrete implementation. We addressed this by establishing a direct mapping. Our P model's state machines and event-driven constructs were designed to mirror the logical flow of the original C++ source code, including explicit modelling of messages, state transitions, and concurrency. Through manual code review in close collaboration with the K2 engineers, we gained confidence in the model's conformance to the code. The tight conformance between the model and the source code also made it easier for engineers familiar with the implementation to identify the root cause of an issue found during verification, as a bug found in P could be directly traced to a specific piece of logic in the source code. The engineers then confirmed the bug's presence in the actual implementation or ruled it out as a false positive, in which case the formal model was refined accordingly.

While automated techniques for model-code conformance validation—such as trace checking [26] and model-based test generation [27, 28]—are an active research area, they require significant engineering investment and face challenges when the abstraction gap between formal models and production code is large. Like most industrial formal methods efforts [3], our close collaboration with the K2 engineers, combined with P's implementation-like semantics, provided practical confidence in model fidelity.

Abstractions. To manage the inherent complexity of a distributed system and focus on the core protocol logic, our formal model employed strategic abstractions. For example, the LSM tree, used in the original implementation to improve I/O performance, was simplified to a key-value map in our formal model. This simplification is valid because the protocol's correctness does not depend on the tree's I/O optimisation, but rather on the eventual consistency of the key-value pairs it manages, which the map preserves.

Furthermore, the treatment of time required special care. K2's protocols rely on a high-precision time clock to generate unique transaction IDs and define epochs. Each primary node uses a local timer to cut epochs, while occasionally querying the timestamp oracle about the true time. The epoch is a critical concept, representing the smallest unit of data to replay on the replica side. Due to P's inability to model real-time or local timers we made multiple abstractions. We used the total number of sent transactions as a proxy for the transaction ID, which guarantees uniqueness. Abstracting the epoch cutting mechanism was more challenging. The key insight was that K2's correctness only depends on the key property that *locally computed epoch durations are always greater than the logical epoch length.* We then came up with a different implementation of epoch cutting in the P model that guarantees the key property, while not restricting the model in any other way: the P model of timestamp oracle periodically broadcasts epoch information to all primary nodes, and on receiving the broadcast, each node queries the timestamp oracle again to get a definitive epoch end time.

Verifying the Protocols. Using the abstractions described above, we constructed a formal model of K2-MVTO representing primary and replica nodes and their interactions, including cross-node transaction replication and replay. In this

[1] Consistency can have weaker definitions too, e.g. *monotonic reads*, *read committed*, *repeatable reads*, *causal consistency*, and *sequential consistency*, based on established consistency models, such as [23] and [24].

model we checked properties including atomicity and consistency. For K2-VCR specifically, we checked monotonicity and primary-backup consistency. The robustness of the protocols was systematically verified by injecting fault scenarios. This involved randomly choosing a node to fail and randomly introducing transaction or request timeouts to ensure the system's resilience under adverse conditions. Through this process, we discovered one bug in fault-free scenarios and one bug in a fault scenario.

3.2 Modelling and Verification: TLA⁺ for K2-MVTO

For this modelling work we took a property-backwards approach. We started by formalising the desired properties in TLA⁺ to gain clarity on the essential process-local and global variables that needed to be captured to verify the property. Listing 1 showcases the TLA⁺ invariants for atomicity, strong consistency, and only monotonic reads. The latter is reported as an example of additional (weaker) consistency level. More invariants were implemented in the model that will not be included in this paper for conciseness.

```
1  Atomicity == \A txn \in 1..TOT_TXNS:
2    /\ Finalized(txn) /\ Committed(txn)
3      => \A k \in KEYS: KeyInTxn(k, txn)
4        => KeyWrittenToTable(k,txn)
5    /\ Finalized(txn) /\ Aborted(txn)
6      => \A op \in WRITE_OPS: OpInTxn(op, txn)
7        => OpNotWrittenToTable(op)
8
9  OnlyMonotonicReads == \A r1, r2 \in readData:
10   /\ r1.k = r2.k
11   /\ r2.reqTT > r1.reqTT
12     => r2.valTT >= r1.valTT
13
14 StrongConsistency == \A t_r \in DOMAIN history:
15   history[t_r].type = "r"
16     => \E t_lw \in 1..t_r :
17       /\ history[t_lw].type = "w"
18       /\ history[t_lw].k = history[t_r].k
19       /\ history[t_lw].v = history[t_r].v
20       /\ ~\E t_ow \in (t_lw + 1)..t_r :
21         /\ history[t_ow].type = "w"
22         /\ history[t_ow].k = history[t_r].k
23         /\ history[t_ow].v # history[t_r].v
```

Listing 1. Snippet of TLA⁺'s state invariants—checked against the K2-MVTO model—for *Atomicity*, *Only Monotonic Reads*, and *Strong Consistency*. The variables `reqTT` and `valTT` contain the timestamp of the read request and the timestamp of the value that was read. So `r2.valTT >= r1.valTT` means that `r2` reads a value that is at least as fresh as what is read by `r1`. The `history` variable is a model artifact representing the global, externally-observable sequence of all operations. The field `type` in history entries distinguishes writes (`"w"`) from reads (`"r"`).

After we identified the target invariants, we built the model around these, making sure it covered the logic that influences the variables checked in the invariants. As an example, we invite the reader to focus on variable `history` in the invariant for strong consistency. The history variable is a global variable, artefact of the model and not present in the design nor implementation, representing the global, externally-observable history of operations in the database. Global variables like this one are key in checking system-wide properties of distributed systems.

To develop a precise TLA⁺ model of the K2-MVTO protocol, we complemented the onboarding material provided by the K2 team with a deep dive in the implementation code. Our goal was to incorporate enough level of detail to build a full *executable design* of the protocol. We started from the entry points to the APIs for `beginTxn`, `write`, `read`, `commitTxn`, and `finalizeTxn`. We isolated the core data structures: `TxnRecord`, `WriteIntent`, and `ReadTracker`. We followed downstream calls, modelling all operations that would manipulate the core data structures and global variables. This process of developing the TLA⁺ model alongside the implementation helped us ensure code conformance and traceability, guaranteeing that the model reflects not only the protocol design but also its implementation and that any bugs found on the model would likely extend to the implementation.

As expected, many modelling and abstraction choices had to be made to avoid getting swerved by implementation-level complexities and instead concentrate solely on the core protocol logic. Network calls were modelled via a shared global variable (the `inbox`). Local concurrent handling of multiple incoming requests was modelled via duplicated event handlers where necessary. This can still introduce potential deadlocks in TLA⁺ when events arrive at states not expecting them. However, these deadlocks are modelling artifacts stemming from TLA⁺'s interleaving semantics: the actual C++ implementation, built on the Seastar framework, spawns concurrent coroutines to handle incoming events in parallel, so the blocking scenarios explored by TLA⁺ cannot occur in practice. Balancing model accuracy and verification tractability was challenging. K2-MVTO is a complex protocol relying on many states, state transitions, and data structures to keep track of the distributed transaction's lifecycle. To aid verification, we used external scripts that automatically generate and run TLC configuration files for progressively increasing bounds, and a machine with multiple physical cores as invariant-checking in TLC benefits tremendously from parallelisation.

3.3 Modelling and Verification: TLA⁺ for K2-VCR

In designing the TLA⁺ model for K2-VCR, we aimed at a higher level of abstraction than in the model for K2-MVTO. To achieve this we made two main design choices.

First, the TLA⁺ model would be based on the high-level description of K2 as presented in the academic paper [6], and not directly mirror the code implementation. As the core property proved about K2-VCR in [6] is a form of read monotonicity (see Eq. (3)), the level of granularity of the TLA⁺ model would be chosen such that the concepts in Eq. (3) and its proof could be expressed.

Second, even though K2-VCR depends on the transaction protocol K2-MVTO, we would treat K2-MVTO as a black-box. This abstraction is valid because the dependency is one-directional: the replica reader only depends on the write stream for epoch information, but neither the write stream nor the overall transaction protocol depend on the replication protocol. Therefore, it seemed possible to model the relevant parts of K2-VCR by looking at only one replica reader and one write stream, where the write stream behaves 'as if' its values were generated by the interaction with primaries according to K2-MVTO. This approach follows the well-known modular paradigm of 'assume-guarantee reasoning' [29]: we want to guarantee that one part of the system, the replica reader, satisfies the desired property Eq. (3) assuming that another part, the write stream, satisfies some other property that is crucial for the replica reader's functioning. To this end, we needed to find the property exhibited by the write stream component in K2 that allowed the proof of Eq. (3) to go through. We found this in the following statement about epoch assignment:

> *The real-time endpoint of the epoch assigned to a transaction is always smaller than the transaction's real-time endpoint.* (4)

Knowing K2's epoch assignment mechanism it is easy to see that transactions in the write stream satisfy property Eq. (4). Our TLA⁺ model then featured only two components: a replica reader that sends arbitrary read requests to a write stream, and said write stream, whose values were non-deterministically updated with entries obeying Eq. (4). In this simple model we verified the read monotonicity property Eq. (3) using TLC. As a sanity check, we observed that if condition Eq. (4) was not enforced in the writestream component, TLC found an error trace violating Eq. (3).

3.4 Key Findings

Verifying K2-MVTO in P highlighted two atomicity bugs: one in the fault-free scenario and one in a fault scenario.

The first atomicity violation (**B1**) would happen when a user wishes to commit a transaction after this has been forced aborted and some of its write-intents have been cleared. Not all the writes are cleared since the TRH is not aware of all of them before receiving the user's commit (or abort) request. The variable marking whether the transaction should be committed is wrongly updated, leading to the remaining in-progress writes to be committed. This bug was fixed by restructuring the logic that stores the commit/abort intent, removing the variable, and simplifying the protocol. The second atomicity violation (**B2**) would happen when a participant node detects a timed-out write operation after the corresponding transaction has been committed but not fully finalised. The node would then request the TRH to abort the overall transaction, the TRH would agree since it erroneously only checks whether the `finalize_action` field of the TxnRecord is set to `None`, leading to the write operations being aborted at the current participant while the others would commit successfully. This bug was fixed by an implementation simplification: the TRH would now return a response based on the TxnRecord state instead of the `finalize_action` field—since it is forbidden to convert a committed state to an aborted state.

Verifying K2-MVTO in TLA⁺ highlighted two atomicity bugs and one violation of the expected consistency level.

The atomicity violation (**B3**) would happen due to an overly strict assumption on how users would invoke the async API. When clients send commit requests without waiting for all writes to be completed, finalisation could be triggered before all write-intents were processed, leading to a transaction being *only partially* written to table (see Fig. 2). The discovery of this atomicity violation increased our understanding of the deep implications of certain assumptions, which led to the decision of introducing an implementation-level enforcement of wait on the client side under certain conditions.

The fourth atomicity violation (**B4**) would happen due to a defect causing transactions to be marked as finalised by the TRH node too early. When this happened, any glitch preventing some of the nodes involved from finalising their write-intents would not be detected by the TRH, and finalisation would not be retried. This bug had a root cause that was similar to that of bug B3 and was fixed by strengthening the finalisation mechanism. The consistency violation (**B5**) caused a TLA⁺ invariant for Read-Your-Writes (RYW) consistency to fail. However, the K2 team later clarified that they did not intend to support RYW consistency in the first release of the database; therefore, the failure was not concerning.

We remark that the TLA⁺ modelling for K2-MVTO started when fixes for **B1** and **B2** were already understood and implemented; therefore these bugs were not found by TLA⁺—but could be reproduced. The new bugs **B3**, **B4**, and **B5** found by TLA⁺ were not found by P because the P model did not cover async API calls and was developed around different assumptions. No bugs were found for the K2-VCR protocol; neither with P nor with TLA⁺.

3.5 Team Composition and Resource Allocation

This project was supported by three engineers from the K2 team, who globally invested approx. one person-month on sharing domain knowledge, advising, and reviewing findings. The P model for both K2-MVTO and K2-VCR was developed

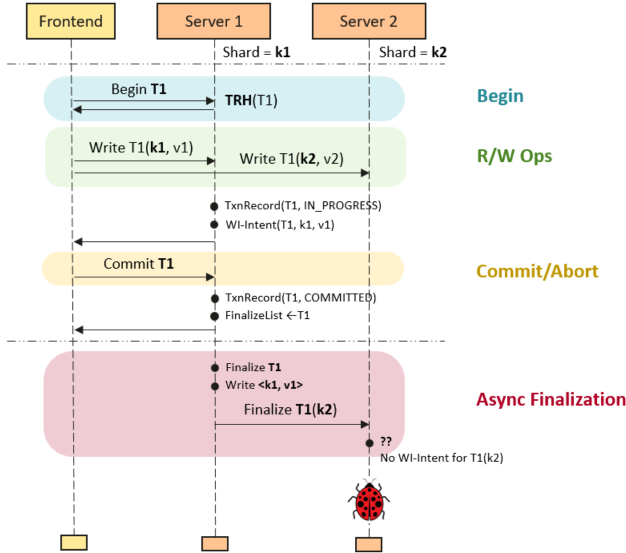

Figure 2. The atomicity violation triggered by bug **B3**. The FrontEnd invokes the Async API to `Begin` transaction T1; sends a `Write` request; and sends the `Commit` request without waiting for acknowledgement of the writes. Server 1, acting as the Transaction Record Holder (TRH), processes both the `Write` and `Commit` requests faster than Server 2, which lags behind. When Server 1 forwards a `Finalize` request to Server 2 for the pending write intents that it holds for T1, Server 2 cannot finalize them because it has not yet processed the `Write` request. A flaw in the finalization retry logic leads Server 1 to write its write intents to storage without retrying at Server 2—violating atomicity.

by two engineers with no prior knowledge in formal methods and P, and limited knowledge on cloud and distributed systems. The TLA$^+$ model for K2-MVTO was developed by one engineer with formal methods expertise, high TLA$^+$ proficiency, and cloud and distributed systems knowledge. Following a two-month period of collaboration to understand the system and its proper abstraction, the actual P modelling was completed in approximately one person-month and the actual TLA$^+$ modelling in 2 person-week. The TLA$^+$ model for K2-VCR was developed at a later time, by one engineer with formal methods background, intermediate TLA$^+$ familiarity, and no prior knowledge on cloud and distributed systems. They were assisted by the engineers who wrote the previous models, as well as by one K2 team member. The project took 2 months until completion.

4 Formal Methods for GSLB's P2P Datastore

The GSLB P2P datastore supports critical distributed services. During the project, a hidden reliability issue in the bonding protocol was uncovered, showing that formal modelling of core behaviours can reveal subtle issues and boost confidence in system reliability under rare failures.

Context and Motivation. Previously, the P2P datastore relied on unit tests, integration tests, and staged deployments, which couldn't fully ensure correctness under complex distributed conditions like gossiping, bonding, or topology changes. Concerns about concurrency, network delays, and timing variations led to adopting formal methods to systematically assess core protocol behaviours alongside the existing tests.

Verification Goal and Properties. This project aimed to increase confidence in the correctness and reliability of the GSLB P2P datastore by formally verifying key distributed behaviours. The main properties for verification (formalized in Fig. 4) were based on system requirements and stakeholder concerns, including:

1. *Bonding eventual consistency*: ensuring that when a peer initiates a bond with another, both eventually agree on the bond's running status, avoiding indefinite mismatches.
2. *Topology connectivity and reachability*: verifying that the mesh network maintains global connectivity, preventing isolated peers or islands.
3. *Gossiping eventual consistency*: guaranteeing that updates disseminated through triplet-based gossip reach all peers within the island in a timely and consistent manner.

Project Onboarding. The GSLB P2P datastore is mainly written in Go and comprises roughly 30 000 lines of code, including about 3 000 comments. It relies on several internal and external dependencies, such as gRPC for remote procedure calls, context-based timeout handling, and custom concurrency utilities. Advanced Go features like goroutines and channels are widely used.

To prepare for verification, we reviewed design documents, architecture and sequence diagrams, production code, and operational slides, and met with the development team to clarify assumptions and highlight critical behaviours. The system was analysed to identify core modules, map peer interactions, and trace execution paths in typical and edge-case scenarios.

4.1 Modelling and Verification: P for P2P

Our initial approach focused on leveraging Gobra. There, bottom-up and code-centric exploration aimed to verify protocol behaviours directly from the implementation, starting with core components such as the RPC layer and gradually extending to more complex repetitive RPC scenarios. While work with Gobra clarified the system's concurrency and consistency patterns, it proved unsuitable for verifying global protocol-level properties within our timeline. This was due to codebase complexity, extensive library dependencies, and various unsupported Go constructs.

We therefore shifted to higher-level modelling with P, which balanced expressiveness and tractability, supported

asynchronous distributed behaviours, built-in concurrency, and systematic exploration of interleavings.

Work Steps and Plan Evolution. The modelling process distilled the system to its core distributed behaviours: mesh construction, triplet gossiping, and island discovery. They were captured in P using abstractions that kept the state space manageable. To ensure alignment with the code, we regularly reviewed the model with the developers and made changes as necessary. The process thus became increasingly iterative rather than linear, with just a few iterations: instead of *plan* ⟶ *model* ⇄ *verify*, the workflow evolved into *plan* ⟶ *(model/abstract* ⇄ *check code-compliance)* ⇄ *verify* with repeated feedback loops.

Challenges and Adaptations. Three main difficulties shaped the modelling effort. First, beyond the state-space explosion, the system exhibited deep execution traces and complex interactions between state machines, especially when modelling goroutines, peer-level timeouts, and realistic peer counts. Second, the production codebase contained implicit concurrency assumptions and undocumented behaviours, which required significant reverse-engineering. Third, protocol-specific constructs, such as gossip-style dissemination, RPCs, and retry loops, were harder to capture in P than expected. These challenges made it necessary to introduce deliberate trade-offs: the model prioritised tractability and correctness reasoning over exhaustive fidelity. In practice, certain code was omitted from the formalisation when it had little impact on core protocol correctness. Examples include logging logic, mutexes, and local goroutine flows. The latter were abstracted by folding bond-associated goroutines into the peer itself, eliminating the need to model fine-grained mutual exclusion or local interactions.

Code/model mappings. A key technical challenge was the translation of Go-specific constructs into P. RPCs and periodic goroutines were mapped into P's event-driven model to preserve semantics while reducing concurrency complexity. Goroutines associated with a peer's bonds were abstracted into the peer itself. For example, gRPC Remote Procedure Calls in the GSLB P2P datastore were translated to show how P mimics synchronous communication while maintaining the original Go semantics [30].

In Go, a client performs synchronous remote calls, automatically handling timeouts, retries, and blocking. If a response takes too long, the call is automatically cancelled, ensuring the program does not hang. In P, the same interactions are explicitly modelled: the client waits for a reply in a blocking loop, but a timeout mechanism is included to avoid deadlocks and ensure progress. Unlike Go, where incoming requests are handled automatically, in P each possible request must be explicitly defined and accounted for, allowing precise control over concurrency and systematic exploration of all possible interactions.

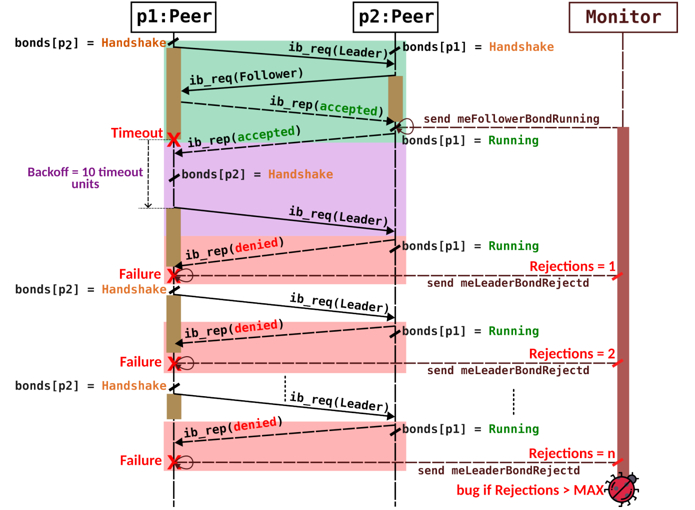

Figure 3. Reliability issue (**B6**) affecting *bonding eventual consistency* property formalized in Eq. (5). The handshake messages `ib_req` and `ib_rep` denote the initiate bond request and reply, respectively.

4.2 Key Findings

While modelling the P2P datastore in P, we identified a previously unknown reliability issue (**B6**) affecting *bonding eventual consistency* requirement (see Fig. 3). In (**B6**), a leader's bond may remain pending while the follower already considers it running, causing unpredictable delays in achieving uniform synchronisation. This behaviour results from interactions between peer optimisers and the bonding mechanism, where timeouts can trigger a follower state that rejects subsequent leader bonding requests following a backoff delay. Consequently, the leader may mark the bond as failed while the follower keeps it as running, leading to repeated retries and prolonged incomplete bonding. Resolving the issue requires either promptly breaking the follower bond to restart the process or implementing a corrective action that accepts the leader's retry. This issue was found by building a P monitor that tracks follower bonds, leader bonds, and rejected leader requests (Fig. 3, right). Activating a follower bond places the monitor in a `hot` state, requiring eventual progress to a leader bond; staying there signals a potential violation. Rejected requests are counted, and exceeding a threshold raises an assertion failure. In terms of the formalization in Fig. 4, the monitor satisfies Eq. (5) while revealing that potentially unpredictable delays might occur between $R(p_f, p_l)$ and $R(p_l, p_f)$.

The *gossip eventual consistency* property was verified successfully under the assumption that key versions are bounded by a ceiling MaxV, to ensure decidability and prevent infinite traces. The property asserts that all peers p must eventually synchronize their local data for every key k to this terminal

$$\forall p_l, p_f \in \text{Peers} : \Box(R(p_f, p_l) \rightarrow \Diamond R(p_l, p_f)) \quad (5)$$

$$\Box\Diamond(\forall p_1, p_2 \in \text{Peers} : p_2 \text{ is reachable from } p_1) \quad (6)$$

$$\Diamond\Box(\forall p \in \text{Peers}, \forall k \in \text{Keys} : \texttt{data}[p][k] = \text{MaxV}) \quad (7)$$

Figure 4. Formalizations of *Bonding eventual consistency*, *Topology connectivity* and *Gossiping eventual consistency* that were used as the basis of P monitors. In Eq. (5), $R(p_f, p_l)$ denotes that a follower peer p_f transitions its bond targeting the leader peer p_l to a *running* state. In Eq. (7), $\texttt{data}[p][k]$ denotes p's local data for key k.

version (see Eq. (7)). The corresponding monitor is shown in Listing 2.

A similar P monitor confirmed *topology reachability*: no isolated islands occur in the modelled scenario.

```
1 spec GossipingConsistency {
2   start state Init { entry { goto Check; } }
3   hot state Check {
4     on meStable do
5     (p: Peer, d_p: map[tKey, (tVer, tValue)]) {
6       data[p] = d_p;
7       if (sizeof(data) == NETWORK_SIZE) {
8         foreach ((p,k) in keys(data, data[p]))
9             assert data[p][k].vr == MaxV;
10        goto Init;
11      }
12      goto Check;
13    }
14  }
15 }
```

Listing 2. The simplified P monitor for Eq. (7) enforces the requirement using a hot state. Upon entering this state, the monitor tracks incoming meStable events, ensuring the protocol avoids stalling in partial synchronization whenever the total count remains below NETWORK_SIZE.

4.3 Team Composition and Resource Allocation

The project was carried out by a single formal-methods engineer, newly hired and with limited prior experience in cloud systems, supported by two GSLB team members familiar with the datastore. The high-level modelling and verification effort in P required about 5 months.

5 Formal Methods for GSLB's Sharder

In this project we used the deductive verifier Gobra to verify the essential functionality of the sharder at the code-level, and found a bug related to the distribution of workload.

Context and Motivation. The sharder library had already been validated through multiple unit tests developed by the engineering teams. At the onset of the formal methods project, it had been operating continuously for approximately three years without any reported bugs. The library combines dense logical reasoning with critical functionality. Because of its compactness we judged the GSLB sharder to be an appropriate target for verification at the code level.

Verification Goal and Properties. The goal of this project was to first identify and then verify crucial correctness properties of the sharding logic.

Project Onboarding. The sharder's logic is contained in a single Go file of roughly 430 lines (including comments) and depends only on Go's standard libraries, not on any other GSLB component. We familiarised ourselves with the sharder by reviewing its well-documented code, examining unit tests, and consulting two development team members. By this time, however, the primary developer of the sharder had left the company.

5.1 Verification: Gobra for the Sharder

Of the few deductive verification tools available for the Go language, we chose Gobra.[2]

```
1 requires acc(tasks)
2 requires forall k int :: {tasks[k].opts} 0 <= k
3   && k < len(tasks) ==> tasks[k].span > 0
4   && tasks[k].subtasks > 0
5   && tasks[k].overlap > 0
6 requires epoch >= 0
7 requires 0 <= chosenTask
8   && chosenTask < len(tasks)
9 requires 0 <= chosenSubtask
10  && chosenSubtask < tasks[chosenTask].subtasks
11 requires chosenSpan == tasks[chosenTask].span
12 // if the epoch satisfies the below equation,
13 // then the task will be assigned
14 requires myMod(chosenTask + epoch, chosenSpan)
15   == myMod(chosenSubtask, chosenSpan)
16 ensures assigned == true
17 func pickSubtasks(tasks []intTask, epoch int,
18   ghost chosenTask int, ghost chosenSubtask int,
19   ghost chosenSpan int)
20   (result []Subtask, ghost assigned bool)
```

Listing 3. Specification of a sharder method. In Gobra, pre- and postconditions are denoted by keywords *requires* and *ensures* respectively. Lines 14-16 correspond to Eq. (9).

Proving a baseline property. The core feature of the sharder is an algorithm that distributes subtasks to workers over the course of various epochs (see Fig. 5a). After familiarising ourselves with the sharder library, we consulted the developers to identify a simple baseline property suitable as the first verification target. We selected the following:

Every subtask is eventually assigned to some worker. (8)

In particular, no task is 'lost' in the assignment. Eq. (8) expresses what is customarily called a *liveness property*. Reasoning about liveness properties involves reasoning about infinite traces (no failure of Eq. (8) could be observed on a

[2] An alternative verifier for go is *Perennial* [31].

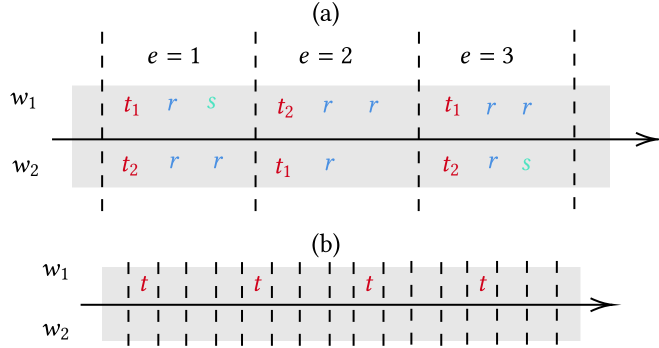

Figure 5. (a) A possible assignment of tasks to workers w_1 and w_2 over the course of three epochs. The first task is composed of two subtasks t_1, t_2 which are configured to run every epoch. r runs three times every epoch, and s runs once per two epochs. (b) A simple instance of the bug (**B7**): w_2 remains unused.

finite trace only) and is considered hard for provers.[3] In fact, verification of liveness properties is currently not supported in Gobra.

At this point, we had taken a deeper look at the sharding logic. In every epoch, the sharder assigns (sub)tasks based on an algorithm that uses modular arithmetic on many variables, including the task index, the number of tasks, the epoch, and general properties of the configuration such as the total number of workers. We were able to extract the following property that underapproximates the set of all epochs in which a task is assigned and serves as a constructive counterpart to Eq. (8):

A subtasks s is assigned to some worker in every epoch e satisfying the equation $(t_s + e) \% p_s = l_s \% p_s$. (9)

Here % is the modulo operator, and t_s, p_s and l_s are some constants related to s that can be inferred from the initial configuration of the sharder.[4] Details of the arithmetic aside, the crucial point is that there are infinitely many solutions in e to Eq. (9). As a consequence, at every point in time there is a future epoch e in which the equation holds, and therefore the subtask is assigned. From this it follows that Eq. (8) holds.

Unlike Eq. (8), Eq. (9) is a so-called *safety property*, meaning that its failure could be observed on finite traces. Safety properties are routinely proved via inductive reasoning, by demonstrating that an invariant holds for all reachable states. Indeed we were able to carry out a formal proof of Eq. (9) in Gobra (see Listing 3). In doing so we achieved full confidence in the baseline property Eq. (8).

[3] Citing [32]: *"While safety can easily be checked by reachability analysis, and many efficient checkers for safety properties exist, more sophisticated algorithms have always been considered to be necessary for checking liveness."*
[4] t_s, is the index of the task that s belongs to, p_s is its span, and l_s is the index of s in its task

As a side remark, while we did not formalize the proof of (9) ⇒ (8), this could be done in principle using a proof assistant that is stronger than Gobra.[5]

A bug. We then looked at developer-written tests to isolate further verification targets. We found that multiple tests aimed to establish that the sharder divides the workload evenly among the workers. We came up with several candidate formalisations of this property and began by attempting pen-and-paper proofs as an initial step. We were unsuccessful in doing this, and eventually found a counterexample. Specifically, we isolated a class of configurations for which the sharder will assign all work to a *single* worker, whereas all other workers remain idle (see Fig. 5b). One worker thus becomes a single point of failure: if it goes down, no work is done at all. This observation was confirmed as a bug by the developers.

The isolated class of configurations is intuitively small (albeit infinite) and also *sparse* within the space of all possible configurations: if one configuration of the sharder exhibits the bug, a minor change to the input configuration will usually lead to normal behaviour of the sharder, meaning that all workers are used. For illustration, one necessary constraint for the bug to occur is that the span of some task is divisible by the square of the total number of workers. Any test can only cover finitely many configurations, and the bug was not caught by the developer-written tests as they simply did not cover the problematic configuration.

A challenge. We exemplarily discuss one technical problem we encountered while working on this verification project, and how we overcame it.

The implementation of sharder makes heavy use of modular arithmetic, and this was also reflected in our formal specifications. Gobra has a built-in operator % for modular arithmetic. We observed however that verifying specifications with % was very slow at best, and non-terminating at worst. In the end we were successful using an *uninterpreted function* `myMod` instead of %. An uninterpreted function is a kind of body-less function given only by its type. By stating additional semantic properties of the uninterpreted function where needed, one can guide the solver to only use these properties when proving some desired step, instead of 'getting lost' using properties that are known to be unproductive.

Fuzz Testing. Some time after conclusion of this project, we went back to see whether the bug could have been found using fuzz testing. We ran a quick check using Go's built-in

[5] One could proceed as follows: Writing $\varphi(s)$ for the statement that some fixed subtask s is assigned to some worker, Eq. (8) becomes the temporal logic formula $(8^*)\Box\Diamond\varphi(s)$. Using another formula $\psi(e)$ that states that modular equation $(t_s + e) \% p_s = l_s \% p_s$ holds, Eq. (9) becomes $(9^*)\Box(\psi(s) \rightarrow \varphi(s))$. By purely arithmetical reasoning, a sufficiently strong prover will be able to show that the modular equation has infinitely many solutions, i.e. $\Box\Diamond\psi(s)$. Together with (9^*) this then derives (8^*).

ID	System	Bug or Issue Description	Severity	Affecting	Phase	Action
B1	K2	*K2-MVTO protocol* A missed internal variable's update caused aborted transactions to be wrongly marked for commit. Bug masked by another method's implementation.	Critical	Atomicity	D/I	Fixed. Protocol simplified by variable removal.
B2	K2	*K2-MVTO protocol* A node moving too fast marked a timed-out operation as aborted clearing its write-intents too early. The other nodes continued execution and committed.	Critical	Atomicity	D/I	Fixed.
B3	K2	*K2-MVTO protocol* Clients using the async API without waiting for acknowledgements before committing could experience atomicity violations and potential loss of data.	Critical	Atomicity	D/I	Addressed by an implementation-level enforced wait on the client side.
B4	K2	*K2-MVTO protocol* In delay scenarios, async finalisation is marked as completed at the TRH despite some nodes not having yet applied their write-intents to table.	Critical	Atomicity	D/I	Fixed.
B5	K2	*K2-MVTO protocol* Read-Your-Write (RYW) consistency is violated and cannot be guaranteed without session ordering.	Minor	Consistency	D/I	Decision was to not support RYWs, aligning with competitors.
B6	GSLB	*P2P protocol* A flaw in the peer bonding logic, preceding the synchronisation phase, caused peers to persist in bonding-retry loop for an unpredictable number of retries.	Minor	Bonding	P	Developers judged the reliability degradation as non concerning. The proposed fix was not implemented.
B7	GSLB	*Sharder library* For some input configurations all work is always assigned to the same worker.	Major	Reliability	P	Pending fix.

Figure 6. Issues found in K2 and GSLB by using formal methods. Atomicity violations are deemed critical as they entail loss of customer data. The *Phase* column indicates whether the issue was found during design/implementation (D/I) or production (P).

(since v1.18) fuzzing capability. The fuzz test found a complicated configuration of the sharder with a very imbalanced distribution of workload, albeit not as bad as the one we observed in the bug. We conjecture that our bug could have been found by analysing the root cause of this example.

5.2 Key Findings

Using Gobra, we were able to prove an essential liveness property of the sharder: every task gets assigned eventually. We also observed one rare bug (**B7**), leading to an extremely unequal division of workload in some configurations.

5.3 Team Composition and Resource Allocation

The project was carried out by a single formal-methods engineer with knowledge of formal methods, but no prior experience with Gobra or the sharder library. They were supported by another engineer with Gobra experience and two developers involved in creating the sharder library. The project took 2.5 months until completion.

6 Synthesis and Lessons Learned

In this section we summarise our findings and share the key lessons we learned. We have organised our observations into ten main takeaways, ordered by how important they are both for teams thinking about adopting formal methods and for teams already using them. The goal is to give such teams a clear sense of what to prioritise based on their own goals, resources, and risk tolerance.

Improved System Correctness and Reliability. This constitutes the immediate short-term ROI of incorporating formal methods in reliability. As summarised in Figure 6, our projects prevented four critical atomicity bugs from entering production; detected one major reliability issue, which is pending a fix; and two minor reliability concerns, which kick-started discussions with stakeholders. As a result of these findings, code quality was improved, the protocols were simplified, and the teams gained a deeper understanding of their systems under different assumptions and edge cases. Notably, the formal modelling and verification of the K2-VCR protocol found no bugs, validating the design improvements made following K2-MVTO's work.

After careful reflection, we think that no other available testing methodology could have found these bugs without a considerable resource investment. The only exception is the bug in the GSLB sharder, which we believe might have been found using fuzzing (see Section 5). Unit and integration tests are not suitable for distributed systems properties and rely on engineers to come up with meaningful test cases, in a process that is inherently biased and susceptible to blind spots. Property-based testing, stress testing, chaos engineering, Jepsen's black-box testing [22], thorough manual investigation, and code reviews might have found some of these bugs (such as K2's consistency issue), but would have required considerable investment in developer team's resources.

Our key takeaway to the reader is that formal methods should be used to target specific categories of deep bugs, where they excel, and should be deployed alongside—not instead of—traditional and advanced testing.

Budgeting for FM Adoption. Fig. 7 lays out metrics from our projects across three key areas: code size, team resources,

	Dev Team			Spec		FM Team				
Project	#	Person-Month	Impl. Lang	LoC	Lang	#	FM Background	Tool Proficiency	Domain Knowledge	Person-Month
K2-{MVTO+VCR}	3	1	C++	4.1k	P	2	-	1	2	5.5
K2-MVTO	3	1	C++	1.4k	TLA⁺/PlusCal	1	✓	4	3	1
K2-VCR	3	1	C++	225	TLA⁺	1	✓	3	1	2
GSLB P2P	2	0.5	Go	4k	P	1	✓	4	2	5
GSLB Sharder	2	0.25	Go	140	Gobra	1	✓	3	1	2.5

Figure 7. Project management metrics. The dev. team support metrics for the three K2 projects are intended global and not repeated—that is, three members of the dev. team supported the K2 verification with its three subprojects, collectively investing 1 person-month. The two members of the FM team who carried out the P modelling for K2 do not have FM background.

Scale	Description	Calibration
1	None	Needs training to do simple tasks with the tool or understand core concepts.
2	Limited	Masters simple tasks but needs support for complex tasks and concepts.
3	Intermediate	Uses main features independently. May struggle with advance functionality and concepts.
4	High	Uses all standard features and resolves most issues without assistance.
5	Expert	Uses all features with ease and mentors others.

Figure 8. Calibration guide for *Tool Proficiency* and *Domain Knowledge*. The assessment of one's tool proficiency (resp. domain knowledge) should consider their familiarity with the underlying formal technique (resp. distributed systems).

and engineer expertise. This gives organisations a concrete reference for estimating FM adoption requirements.

For code metrics, we present the formal model's line count (for P/TLA⁺), resp. the number of assertions (for Gobra), along with the language of the original codebase. We also mention the size and time investment of the stakeholder team—in our case, the developers of the target systems—supporting the FM engineers. The last three columns focus solely on the FM teams's capabilities. We capture two dimensions of their starting point: *tool proficiency*—their prior experience with the specific FM tools (or comparable ones) and understanding of the underlying theory (like state machines for P or temporal logic for TLA⁺), and *domain knowledge*—familiarity with the target system or similar architectures. Engineers self-rated both skills using the scale in Figure 8.

Notably, team size was not a constraint in our work. We found that even small teams—often consisting of just one engineer per project—could effectively carry out formal verification tasks if supported by the development team.

Model Validation. We advocate treating formal model validation as a deliberate, first-class step in FM workflows.

Many safety properties are trivially satisfied by overly restrictive models. For instance, a model that deadlocks or omits key system traces may falsely appear to prove safety properties, despite the actual system being vulnerable to failures. Since formal models are intentional abstractions capturing only variables relevant for protocol correctness, they may omit behaviors present in the real system. It is therefore imperative to validate that the model captures sufficient behavior to verify the intended properties.

Several times we initially believed a model verified certain safety properties, only to later find it excluded traces that could violate those properties. To catch these silent failures early, we intentionally 'broke' models during validation—temporarily disabling safety checks to verify the property could fail under reasonable conditions. If the property holds even when we try to break it, that indicates excessive abstraction or incorrect behaviour encoding.

Finding the Right Abstraction. Choosing the appropriate abstraction involves a tradeoff: highly abstract models minimise state space explosion risk but may become too detached from the codebase. Conversely, code-aligned models ensure stronger confidence but risk state space explosion. A simple balancing method is to write multiple models. We did so in our work on K2, where a P model closely aligned with the codebase was complemented with an abstract TLA⁺ model. The P model was written bottom-up from the codebase, applying abstractions only where necessary. The TLA⁺ model was written top-down, starting from the property formalisation and modelling only crucial system parts.

Domain Knowledge Bottleneck. The primary bottleneck we encountered was the time required for engineers to understand the codebase or protocol under verification. This is to some extent unavoidable in post-hoc verification projects where the verification engineers differ from the original development team. Across our projects, approximately 60% of time was spent on understanding the target system, leaving only 40% for the actual modelling and verification process.

Efficient knowledge transfer between developers and FM engineers serves as a critical solution here, and so we advocate close collaboration. Regular weekly or biweekly checkpoints with the development team proved beneficial in our experience. Given our international setup, where both development and FM teams were spread across locations, having

at least one FM team member located closer to the development team was essential. Embedding an FM team member temporarily within the development team to gain deeper system insights could further reduce onboarding time.

FM Long-Term ROI. FM teams should aim to extend their efforts' benefits beyond short-term project deliverables—whether proof generation or bug discovery—to sustain investment value. Tools like model-based automated test generation can ensure formal models continuously align with evolving codebases, enabling future property validation. Developers may prototype code changes first in the formal model to assess impact in a sandboxed environment. Even models that report no violations further aid fault localisation: when runtime defects emerge later in the actual system, they typically originate in areas intentionally abstracted by the model, such as simplified concurrency or omitted edge cases. This creates focused debugging boundaries.

Long-term reuse requires models modifiable by developers with minimal FM support. We invested in comprehensive documentation and explicitly marking modification points.

Lightweight vs. Heavyweight FM. A crucial initial step in every FM project is the choice of tools.

Lightweight formal methods, such as P and TLA$^+$, allow users to construct abstractions. By carefully selecting these abstractions, it becomes feasible to encapsulate even very large codebases into models of manageable size. Thus, lightweight formal methods can work well in large-scale projects. There are many lightweight model-based projects implemented at companies like Amazon [3], DeepSeek [11], MongoDB [15] and Microsoft [13]. Our experience in verifying K2 confirms once again [3] that engineers without formal methods background can efficiently apply lightweight formal methods after only minimal training.

Heavyweight formal methods that operate at the code level (e.g., deductive verifiers) typically incur effort that is commensurate to the codebase size. This makes them harder to use for full-scale industrial systems, but they remain a good choice for small critical components—such as our task sharder library. A large-scale industry example of applying heavyweight formal methods to critical systems is Amazon's use of the deductive verifier Dafny [33] in proving the correctness of its authorisation engine [34]. Deductive verifiers require understanding of mathematical logic, and we recommend teams without such a background to factor in appropriate time for training.

Tool and Community Support. TLA$^+$ and P have extensive online documentation [35–37] and a low learning curve. Both come with VSCode extensions offering syntax highlighting, automated compilation, and visual error trace debugging. With some prior experience in formal methods, deductive verifiers like Gobra are also approachable. Although less established as P and TLA$^+$, Gobra is supported by online documentation [38, 39], a useful VSCode extension, and a small but active community on Zulip and GitHub. For example, at one point during our work on the task sharder we identified a bug in Gobra's implementation of modular arithmetic on negative numbers. This bug was reported to the developers and quickly fixed [40]. Altogether, we can report that our time learning P, TLA$^+$ and Gobra was only a marginal factor, and we strongly encourage developers of formal methods tools to not take usability lightheartedly.

When Not to Use Formal Methods. It should also be highlighted that not every software project is a reasonable target for formal methods. This applies in particular to projects that have limited reliability requirements—failures do not lead to data loss, bugs have little effect and cheap fixes—or little business impact—think non—critical internal tools, tools with small radius, no customer effect etc. Likewise, not a good target for formal methods are systems for which traditional testing works well. This may be the case if the system's behaviour is deterministic and not influenced by external factors, edge cases are easily reproducible, and unit tests already achieve high coverage.

Actionable Checklist of a Successful FM Project. We condense our experience into the following checklist for the stages of a successful formal methods project:

Pre-modelling/verification: Confirm that the system is a good target for FM • Confirm implementation stability • Agree with stakeholders on target properties for verification • Choose FM tool • Secure ≥3-month commitment from stakeholders • Budget training unless expertise in-house.

In-modelling/verification: Start with the property and build backwards • Define model validation strategy • Validate models early with known scenarios • Document abstraction decisions • Hold regular stakeholders checkpoints.

Post-modelling/verification: Document model and verification conditions • Transfer knowledge • Invite stakeholders to own the model and the specifications • Turn the model into a reusable deliverable.

7 Conclusion

We looked at three formal methods projects targeting components of foundational Huawei Cloud services. Beyond individual project reports, we gathered findings, data, and distilled lessons learned to give practical guidance to teams and organisations considering adopting formal methods in their own development workflows. Our experience shows that formal methods are applied successfully in modern development environments with moderate investment.

Acknowledgements. We would like to acknowledge Dinesh Wijekoon and Blaz Sovdat for their assistance with the sharder and P2P project respectively. We also thank the reviewers for their feedback which improved this paper.

References

[1] Anna Zamansky, Maria Spichkova, Guillermo Rodríguez-Navas, Peter Herrmann, and Jan Olaf Blech. Towards Classification of Lightweight Formal Methods. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018*, pages 305–313. SciTePress, 2018. ISBN 978-989-758-300-1. doi: 10.5220/0006770803050313. URL https://doi.org/10.5220/0006770803050313.

[2] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. Formal Methods in Industry. *Form. Asp. Comput.*, 37(1), December 2024. ISSN 0934-5043. doi: 10.1145/3689374. URL https://doi.org/10.1145/3689374.

[3] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 2015. URL https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods.

[4] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jay Lorch, Oded Padon, and Bryan Parno. Verus: A Practical Foundation for Systems Verification. In *2024 Symposium on Operating Systems Principles*, pages 438–454, November 2024. URL https://www.microsoft.com/en-us/research/publication/verus-a-practical-foundation-for-systems-verification/.

[5] Alastair Reid, Shaked Flur, Luke Church, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are. In *HATRA 2020: Human Aspects of Types and Reasoning Assistants*, 2020. URL https://arxiv.org/abs/2010.16345.

[6] Haoze Song, Yongqi Wang, Xusheng Chen, Hao Feng, Yazhi Feng, Xieyun Fang, Heming Cui, and Linghe Kong. K2: On Optimizing Distributed Transactions in a Multi-region Data Store with True-time Clocks. *Proc. VLDB Endow.*, 18(6):1756–1769, 2025. URL https://www.vldb.org/pvldb/vol18/p1756-song.pdf.

[7] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[8] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013.

[9] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. *ACM SIGARCH Computer Architecture News*, 38(1):167–178, 2010.

[10] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. 2021. URL https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3.

[11] Repository with P specifications. https://github.com/deepseek-ai/3FS/tree/main/specs.

[12] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994. ISSN 0164-0925. doi: 10.1145/177492.177726. URL https://doi.org/10.1145/177492.177726.

[13] Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. Understanding Inconsistency in Azure Cosmos DB with TLA+. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 1–12, 2023. doi: 10.1109/ICSE-SEIP58684.2023.00006.

[14] Markus A. Kuppe, Finn Hackett, and Josh Rowe. Going Beyond an Incident Report with TLA+. *USENIX*, July 2023. URL https://www.usenix.org/sites/default/files/login_-_going_beyond_an_incident_report_with_tla_.pdf.

[15] William Schultz and Murat Demirbas. Design and Modular Verification of Distributed Transactions in MongoDB. *Proc. VLDB Endow.*, 18(12): 5045–5058, September 2025. ISSN 2150-8097. doi: 10.14778/3750601.3750626. URL https://doi.org/10.14778/3750601.3750626.

[16] Xiaosong Gu, Wei Cao, Yicong Zhu, Xuan Song, Yu Huang, and Xiaoxing Ma. Compositional Model Checking of Consensus Protocols Specified in TLA+ via Interaction-Preserving Abstraction. *CoRR*, abs/2202.11385, 2022. URL https://arxiv.org/abs/2202.11385.

[17] Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In *International Conference on Computer Aided Verification*, pages 367–379. Springer, 2021.

[18] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49122-5.

[19] Robert W Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1993.

[20] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[21] J. C. Pereira, T. Klenze, S. Giampietro, M. Limbeck, D. Spiliopoulos, F. A. Wolf, M. Eilers, C. Sprenger, D. Basin, P. Müller, and A. Perrig. Protocols to Code: Formal Verification of a Next-Generation Internet Router. Technical report, 2024. URL https://arxiv.org/abs/2405.06074.

[22] Kyle Kingsbury. Jepsen, 2025. URL https://jepsen.io/.

[23] Kyle Kingsbury. Jepsen Consistency Model, 2025. URL https://jepsen.io/consistency/models.

[24] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D., MIT, Cambridge, MA, USA, March 1999. Also as Technical Report MIT/LCS/TR-786.

[25] ScyllaDB. Seastar, 2025. URL https://seastar.io/.

[26] Heidi Howard, Markus A. Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. Smart casual verification of the confidential consortium framework. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-46-5.

[27] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. Extreme modelling in practice. *Proc. VLDB Endow.*, 13(9):1346–1358, May 2020. ISSN 2150-8097. doi: 10.14778/3397230.3397233. URL https://doi.org/10.14778/3397230.3397233.

[28] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 127–143, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi: 10.1145/3552326.3587442. URL https://doi.org/10.1145/3552326.3587442.

[29] A. Pnueli. *In transition from global to modular temporal reasoning about programs*, page 123–144. Springer-Verlag, Berlin, Heidelberg, 1989. ISBN 0387151818.

[30] gRPC Documentation. https://grpc.io/docs/.

[31] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 243–258, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359632. URL https://doi.org/10.1145/3341301.3359632.

[32] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In Elsevier, editor, *Electronic Notes in Theoretical Computer Science*, 2002.

[33] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[34] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clement Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. Formally Verified Cloud-Scale Authorization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2508–2521, 2025. doi: 10.1109/ICSE55347.2025.00166.

[35] TLA+ webpage. https://lamport.azurewebsites.net/tla/tla.html.

[36] Learn TLA+. https://learntla.com/.

[37] P Language Manual. https://p-org.github.io/P/manualoutline/.

[38] Practical Program Verification in Go with Gobra. https://viperproject.github.io/gobra-book/.

[39] A Tutorial on Gobra. https://github.com/viperproject/gobra/blob/master/docs/tutorial.md.

[40] Gobra Issue 858. https://github.com/viperproject/gobra/issues/858.