Turning an incident report into a design issue with TLA+

Finn Hackett (UBC) & Markus A. Kuppe (MSFT)

By Some Not-SREs

A. Finn Hackett
PhD Student at U. of British Columbia

Programming Languages Researcher
→Intern at Microsoft Research S‘22

Markus A. Kuppe
Principal Research Engineer (MSFT)

TLA+ eng, Postmortem Reviewer
→Finn’s internship mentor S’22

Joshua Rowe
Principal Engineer (MSFT)

Azure CosmosDB
→Domain expert

Thanks to postmortem owner **Ben Pannell** (MSFT)!

Highlights

📉 We learned about a high-profile 28-day incident at Microsoft Azure.

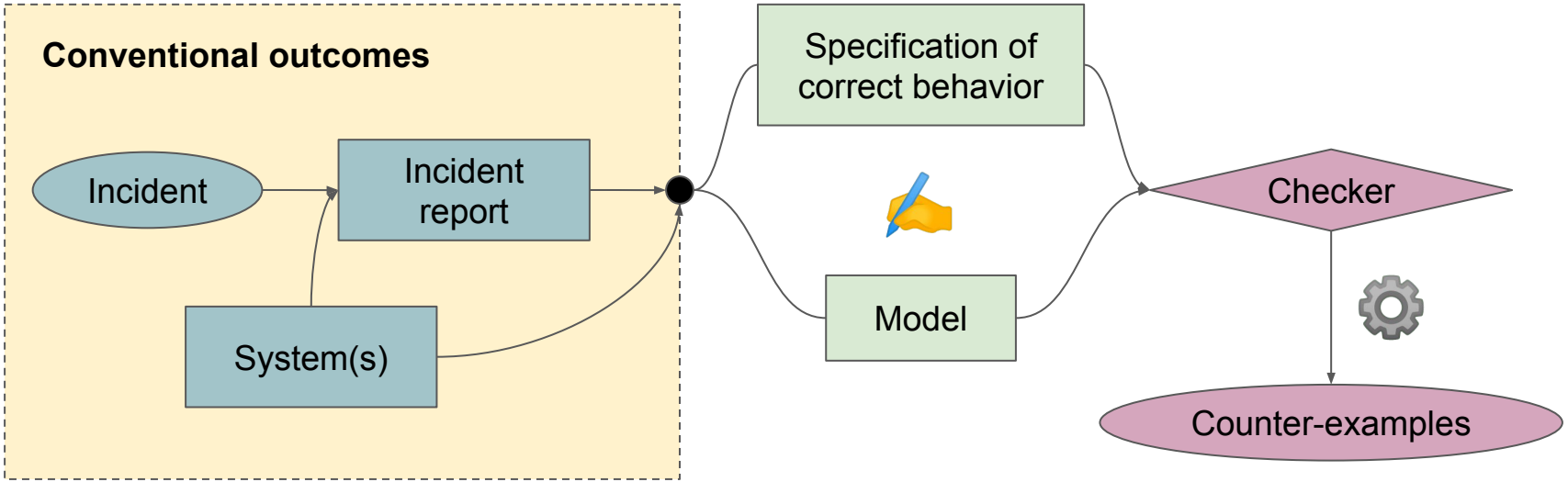
😕 Mitigation was a feature revert (full repair a costly multi-year redesign).

🤔 Reproducing the issue at small scale is impractical…

💡 **Idea: use modeling to unambiguously document the problem.**

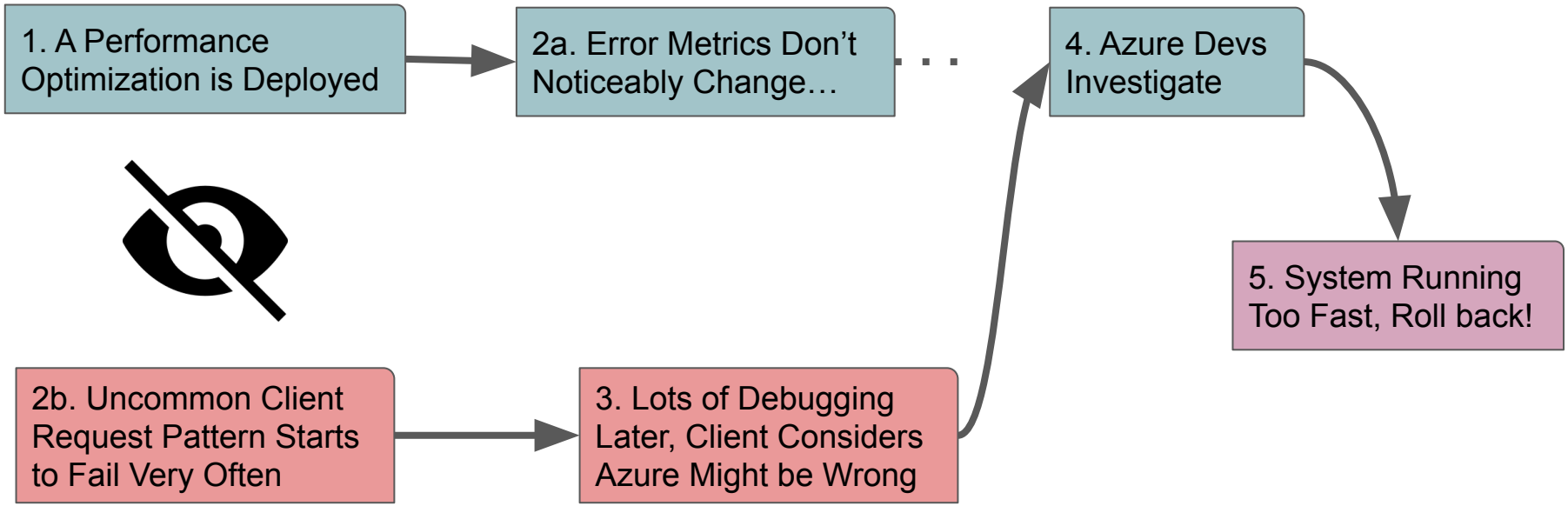
We use TLA+ here, but this applies to similar tools as well.

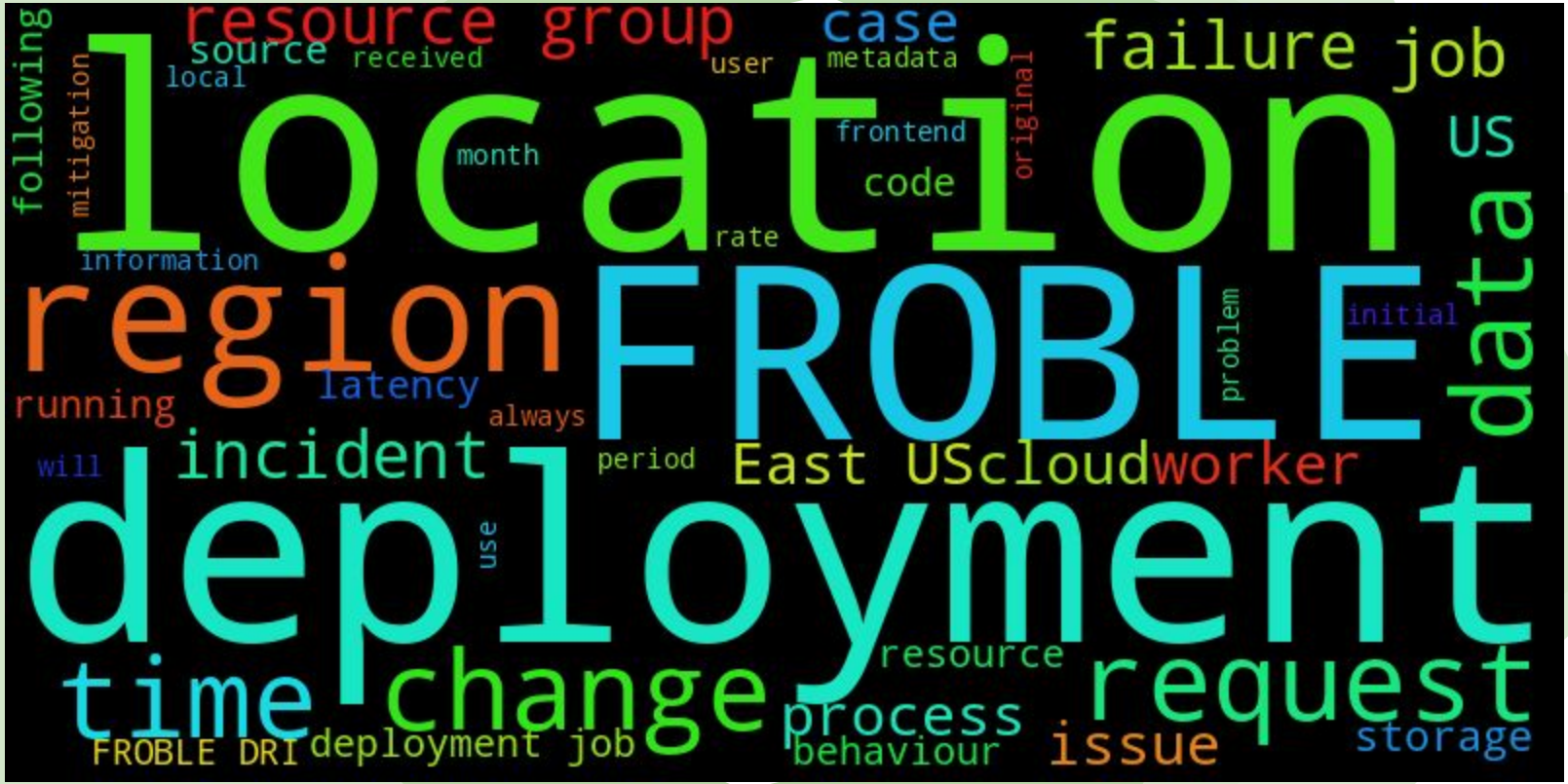
A Workflow Beyond Incident Reports



… which summarize the issue at a high level.

Story Time: an Incident





s/${real product name}/FROBLE

What's the Big Deal?

Client is not seeing errors anymore, but…

😕 That took a long time to figure out.

(28 days to fix)

😕 We know why we rolled back, but design-level insight is missing.

🤔 What about the design made a component being *too fast* a problem?

We want to clearly model and understand what happened at the conceptual level.

Step 1: Understand Underlying System(s)

What is Azure CosmosDB?

⚙️ **Need to understand what the system is supposed to do to model it.**

What we present next is based on what we learned by drafting our model.

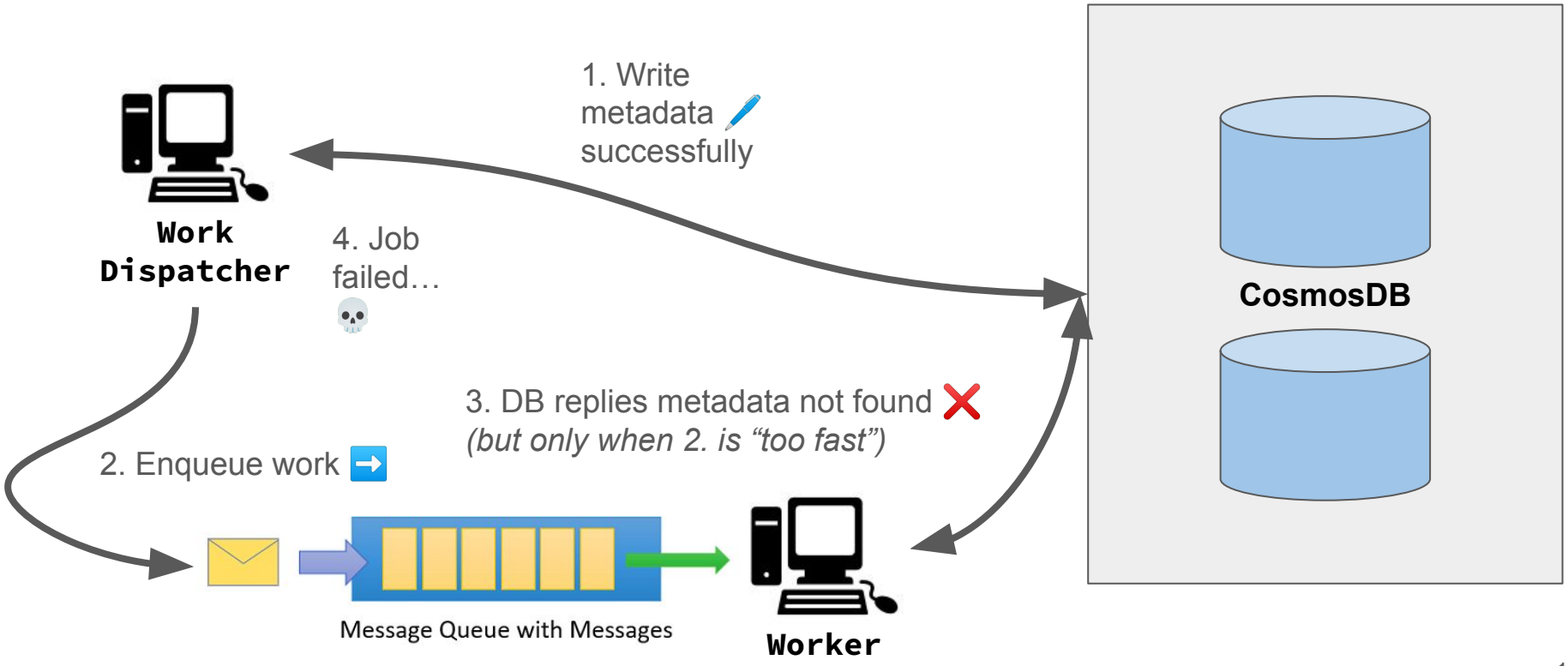
🌎 A planet-scale key-value store - a big distributed system.

🗃️ Stores your data as key-value mappings, useful for app/service data.

🤔 Similar to? Amazon DynamoDB, Google Firestore…

Also using TLA+

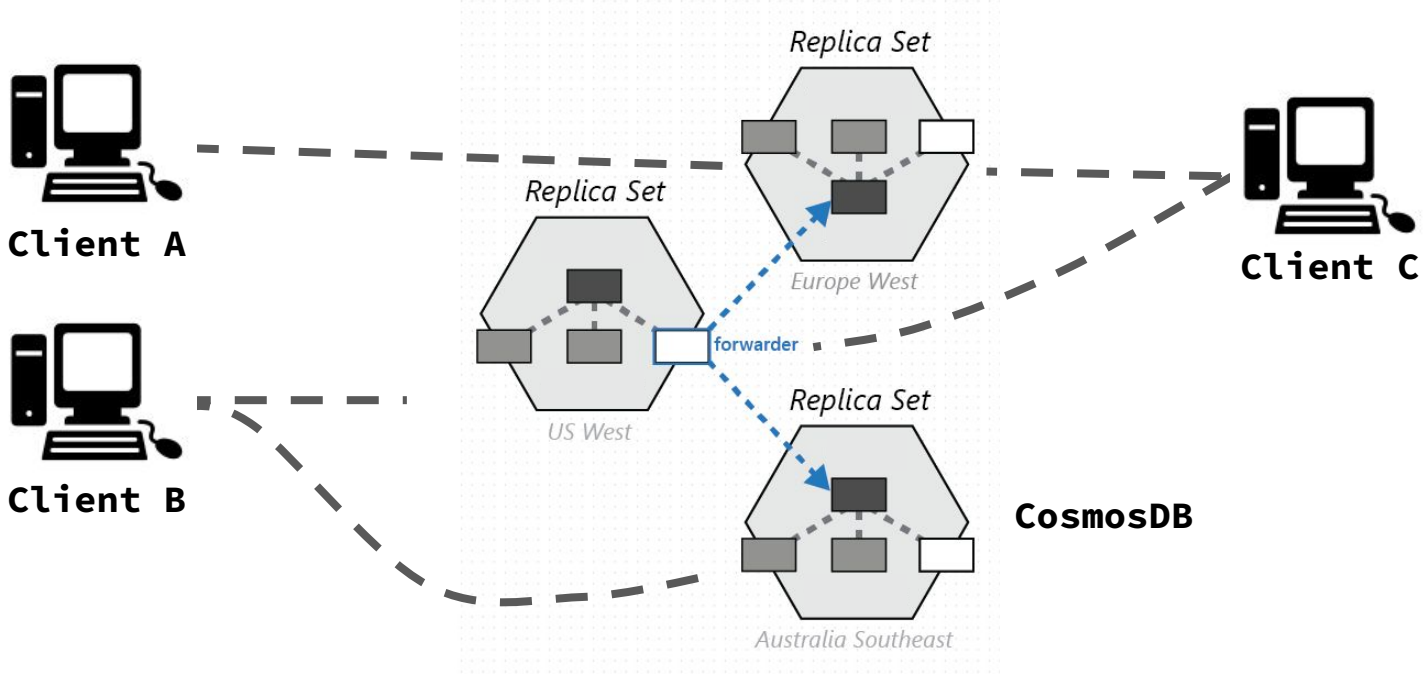
Key Detail: Communication with Azure Cosmos DB



CosmosDB: Consensus Machine

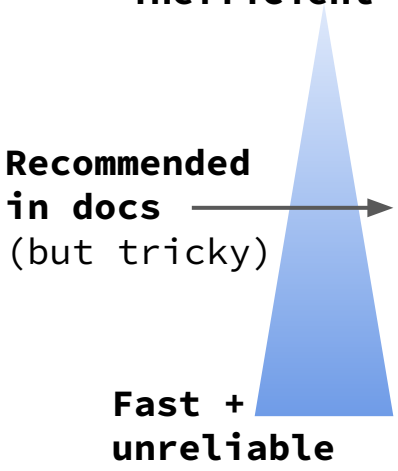
📩 Clients exchange get/set messages with different servers.

👉 Need consensus to make sense of this.



CosmosDB: Consistency Levels

🛠️ 5 consistency levels control how strictly CosmosDB maintains consensus.



Consistency Level	Effect on Client View
Strong Consistency	Global order; clients always see latest versions
Bounded Staleness	Old values visible for limited (*bounded*) time
Session Consistency	Synchronize only between clients with shared token
Consistent Prefix	Generally similar to Eventual Consistency
Eventual Consistency	Old values should eventually stop showing up

Step 2: Represent Underlying System(s)

CosmosDB: What to Model for Our Purpose?

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."
– Edsger W. Dijkstra

No

❌ Detailed client API syntax

❌ Server management

❌ Anything client can't see

Yes

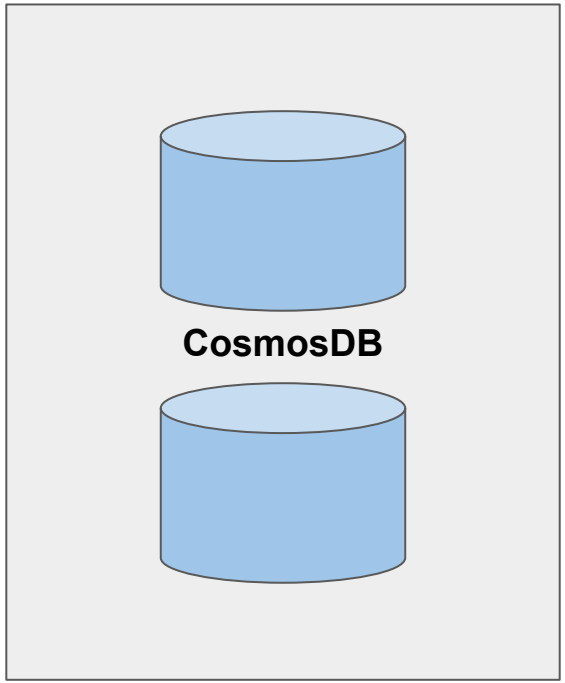
✔ Plain key-value r/w

✔ Client view of servers

✔ Anything client can see

(especially unusual things)

… But Have a Domain Expert Model It



CosmosDB: the TLA+ Model

5✔ Simulate concurrent key-value reads and writes at all 5 consistency lvls.

⌛ Reusable, complete model: took 3 months to build w/ dev input.

🌟 But it took 1 day to use in this postmortem.

Github: https://github.com/tlaplus/azure-cosmos-tla/tree/master/simple-model

Understanding Inconsistency in Azure Cosmos DB with TLA+

Finn Hackett
University of British Columbia
Vancouver, Canada
fhackett@cs.ubc.ca

Joshua Rowe
Microsoft
Redmond, USA
joshua.rowe@microsoft.com

Markus Alexander Kuppe
Microsoft Research
Redmond, USA
makuppe@microsoft.com

***Abstract*—Beyond implementation correctness of a distributed system, it is equally important to understand exactly what users should expect to see from that system. Even if the system itself works as designed, insufficient understanding of its user-observable semantics can cause bugs in its dependencies. By focusing a formal specification effort on precisely defining the expected user-observable behaviors of the Azure Cosmos DB service at Microsoft, we were able to write a formal specification of the database that was significantly smaller and conceptually simpler than any other specification of Cosmos DB, while representing a wider range of valid user-observable behaviors than existing more detailed specifications. Many of the additional behaviors we documented were previously poorly understood outside of the Cosmos DB development team, even informally, leading to data consistency errors in Microsoft products that depend on it. Using this specification, we were able to raise two key issues in Cosmos DB's public-facing documentation, which have since been addressed. We were also able to offer a fundamental solution to a previous high-impact outage within another Azure service that depends on Cosmos DB.**

***Index Terms*—cloud computing, formal methods, model checking, documentation**

I. Introduction

Consistency guarantees for distributed databases are notoriously hard to understand. Not only can distributed systems inherently behave in unexpected and counter-intuitive ways due to internal concurrency and failures, but they can also lull their users into a false sense of functional correctness: most of the time, users of a distributed database will witness a much simpler and more consistent set of behaviors than what is actually possible. Only timeouts, fail-overs, or other rare events will expose the true set of behaviors a user might witness [1]. Testing for these scenarios is difficult at best: reproducing them reliably requires controlling complex concurrency factors, latency variations, and network behaviors. Even just producing usable documentation for developers is fundamentally challenging [2], [3], [4], and explaining these subtle consistency issues via documentation comes as an additional burden to distributed system developers and technical writers alike. Formal methods have long been applied to the design of distributed systems, including in industry [5], [6], [7], [8], but these are years-long high-effort projects that focus on implementation correctness, not explaining the system to users. Rather than focus on this difficult task, we address a simpler and more fundamental question: ignoring the implementation, what kind of behavior *should* a client be able to witness while interacting with a service?

We use TLA+ to answer this simpler question for Cosmos DB, a planet-scale key-value store. In practice, Cosmos DB offers a rich interface featuring multiple query APIs, and has complex operational behaviors involving georeplication and partitioning of data. As our focus is on data consistency from a client perspective, we model only the core read and write operations underlying the system's semantics relating to their 5 configurable consistency levels. We show that this minimal client-focused specification of a large-scale service offers important design- and documentation-level insights, while keeping buy-in cost low.

We document the 2 person-month development process of our specification, which consists of iterative prototyping using the public documentation [9], feedback from author 2, a Cosmos DB developer, and the specification and model checking of a collection of formal properties based on our understanding. Aside from the specification itself, we discuss a pair of key issues it helped us discover within Cosmos DB's documentation, and how both have since been addressed. We also use our specification to explain the previously-unclear root cause of a 28-day high-priority outage within Microsoft Azure.

We describe the following results: (1) a concise (390 LOC) client-focused specification of Cosmos DB, a large-scale distributed system; (2) a pair of key documentation bugs we found by developing our specification — statements in Cosmos DB's public documentation [9] that have now been corrected; and (3) using our specification, a novel and concise mechanized explanation of a high-severity Cosmos DB-related outage within Azure that took 28 days to identify and mitigate.

Beyond our work so far, we expect our specification to be useful in future design work as Cosmos DB's implementation evolves, aided by its ability to precisely and abstractly state a client's expectations of system behavior. Services depending on Cosmos DB may also benefit from incorporating our work into TLA+ specifications of their own processes, in which case our work may be used to prevent future outages similar to the one we describe in this paper.

II. Background

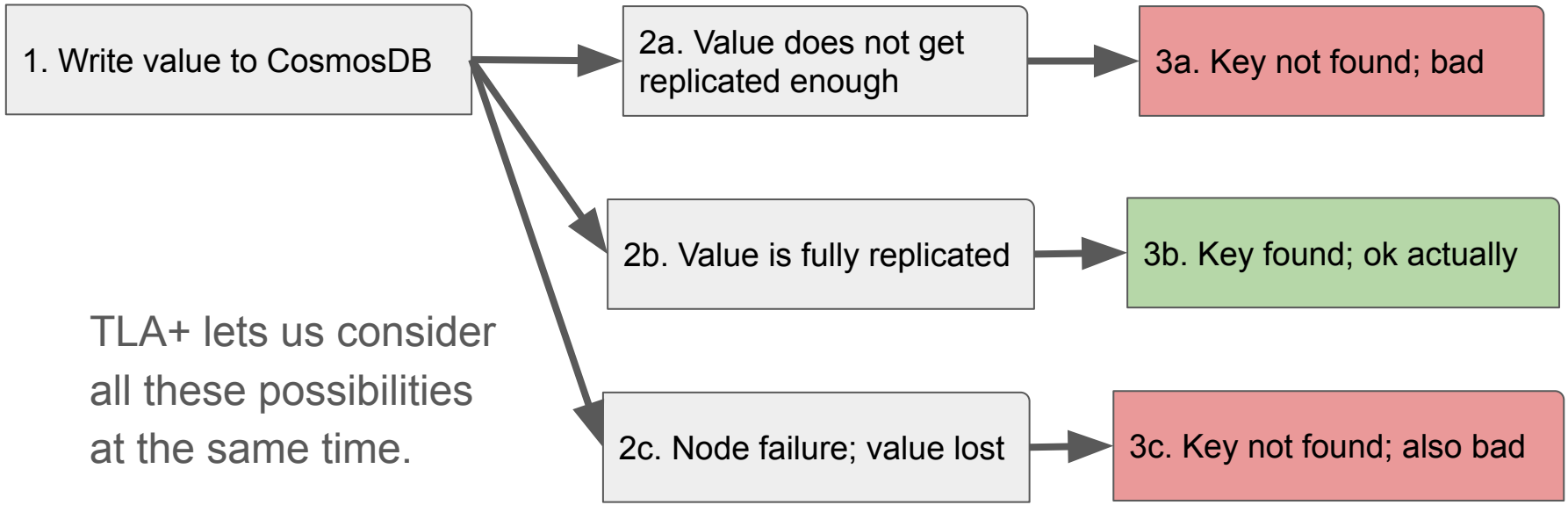
Our work uses the TLA+ specification language [10], which can be used to describe state machines using set-theoretic constructs and temporal logic. Models written in TLA+ have no

See our conference paper for all the details:
https://doi.org/10.48550/arXiv.2210.13661 (preprint)

Step 3: Model the Incident

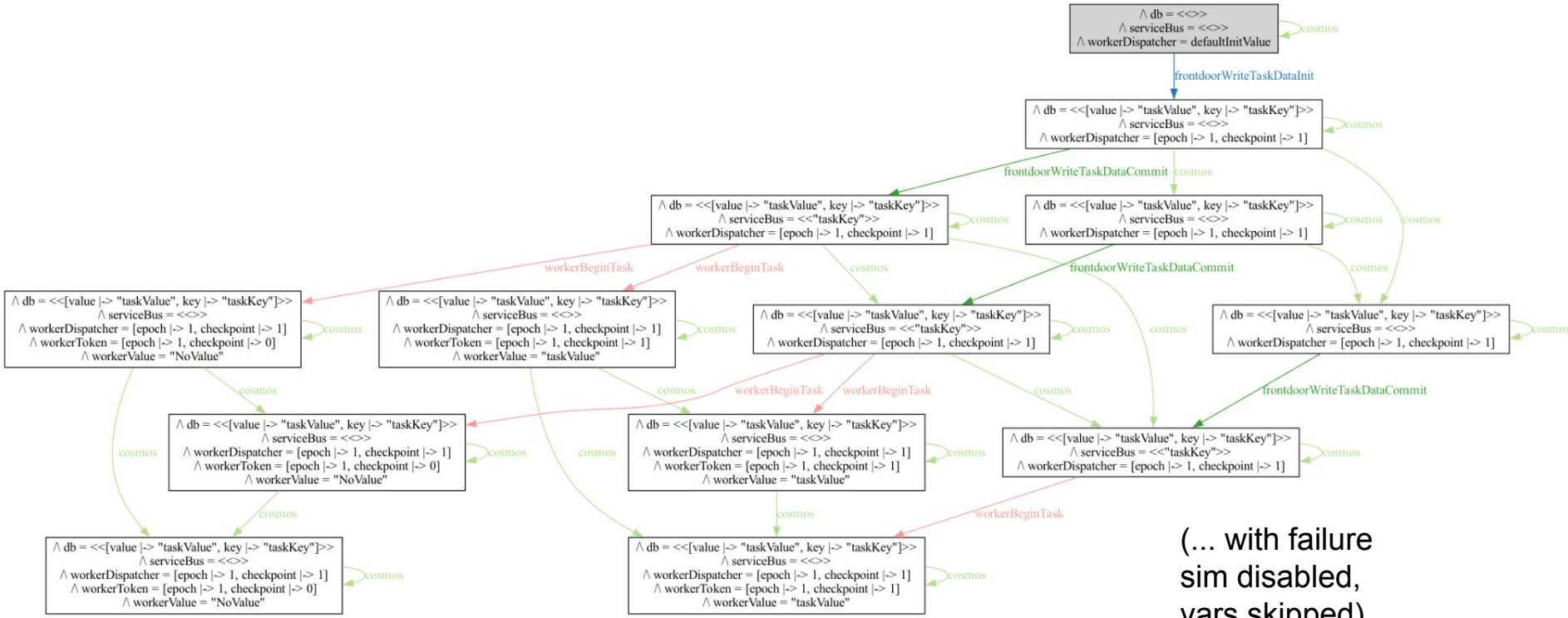
Demo Time!

Thinking in States: Branching Possibilities



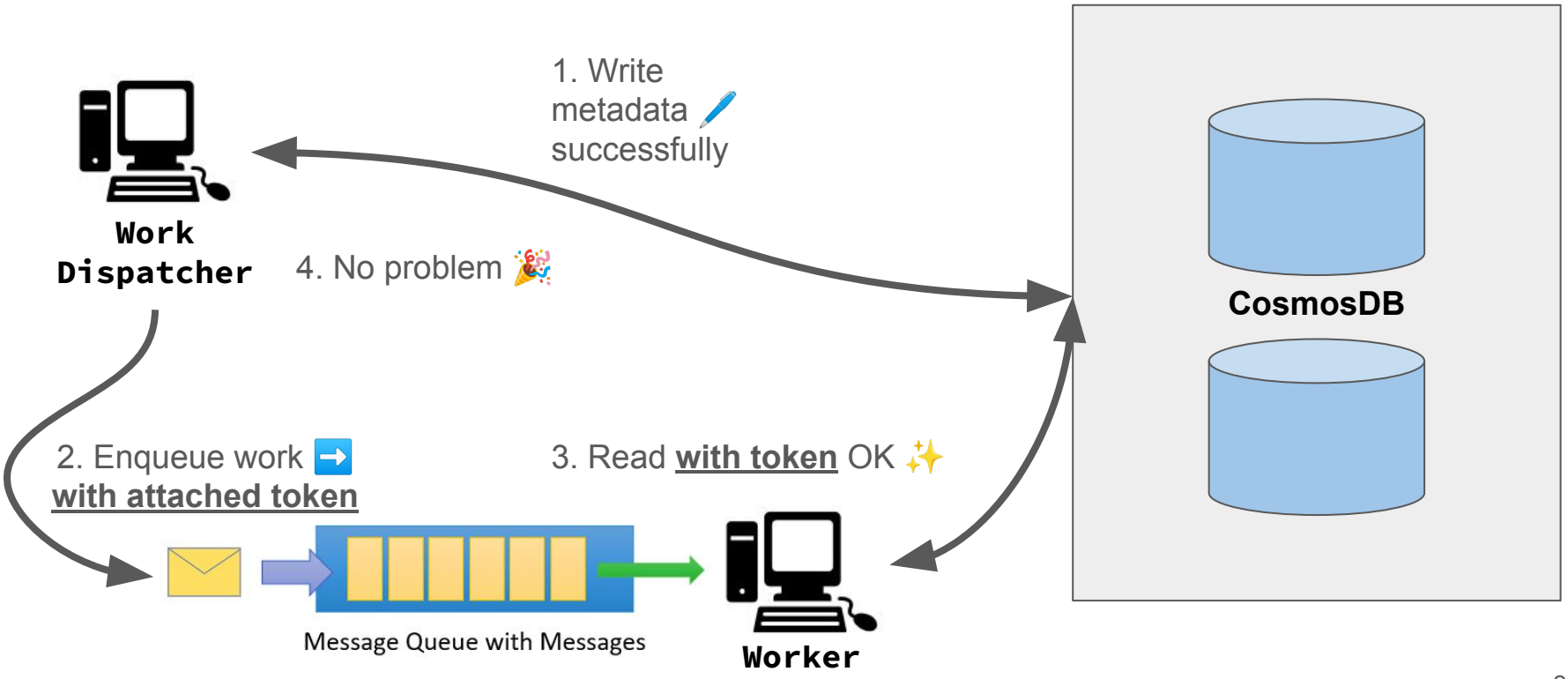
TLA+ lets us consider all these possibilities at the same time.

Thinking in States: the Full 



(... with failure sim disabled, vars skipped)

The Corrected Design



What We Learned

💡 Problem is using session consistency without sharing tokens.

📝 Original mitigation changed the probability of the issue, but wasn't a fix. The issue was always there, just very rare.

📝 Clear when presented this way, but issue was hidden behind APIs.

📝 Engineers were not surprised by our results - we confirmed what was was suspected but could not be demonstrated.

📝 Modeling helped us think and investigate.

Summary

Reproduced incident spanning multiple foundational systems. 🎉

Our modeling workflow can go beyond current incident reports.

You can do this too. Learn at http://tlapl.us.

TLA+ model (linked from official docs):
https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

Paper: https://doi.org/10.48550/arXiv.2210.13661.

Any Questions?