Weeks of Debugging can save you hours of TLA+

...building a ~~distributed~~ cloud system

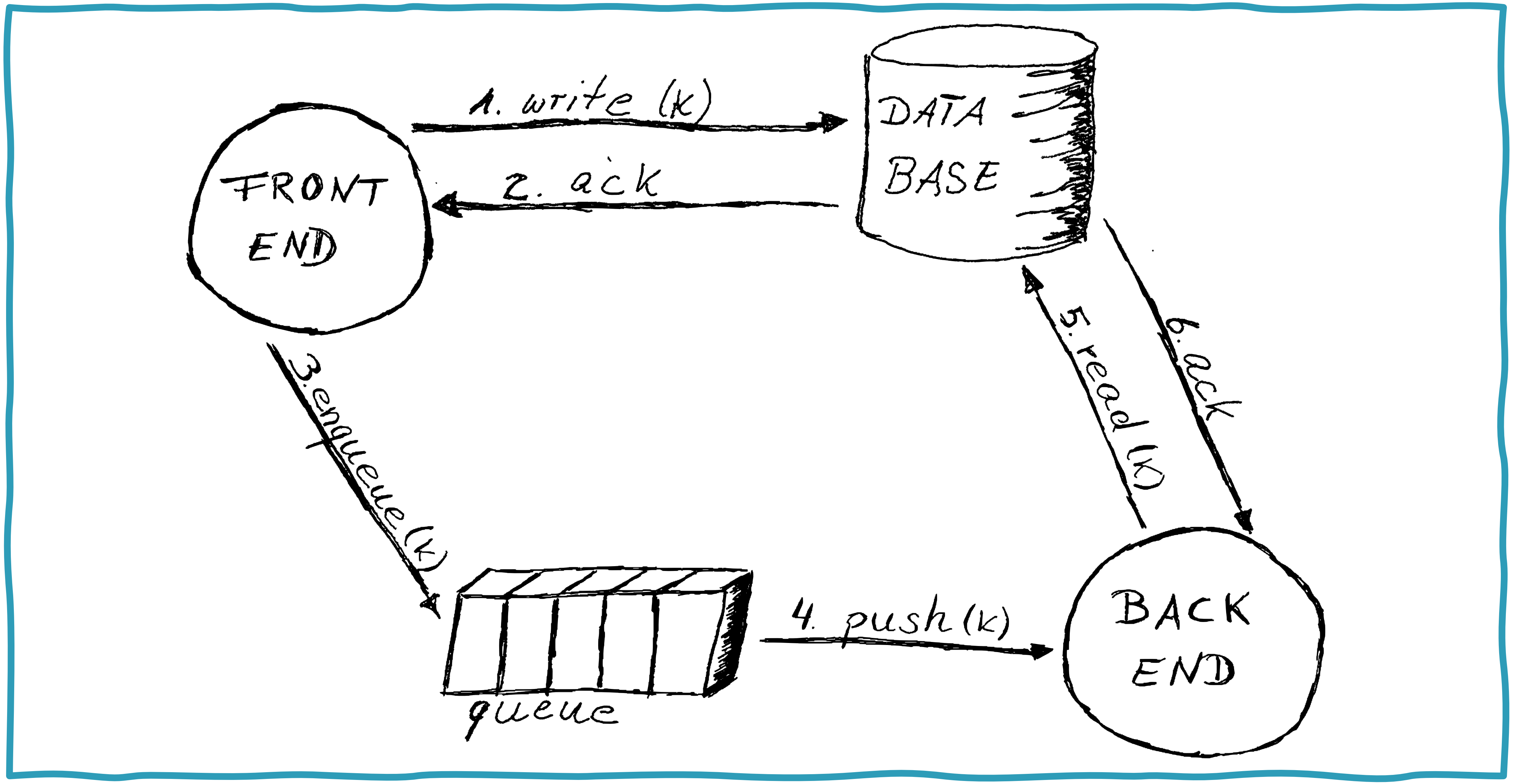
Markus A. Kuppe – Principal Engineer - Microsoft Research

Your average cloud system

- A client (frontend) issuing user requests
- Some database system for persistence
- A message queue/broker to decouple frontend and backend
- A backend system

Your average workflow

1. User places an order
2. Frontend writes the order to the database
3. ACKs the order to the user
4. Frontend publishes a message
5. Backend receives message, reads order from database, and fulfills it



1. write (k)
2. ack
FRONT END
DATA BASE
3.enqueue (k)
queue
4. push (k)
BACK END
5. read (k)
6. ack

@lemmster

Let's build our first prototype

Get a subscription for your favorite cloud provider

Set up infrastructure: frontend, database, queue, backend server

Implement and deploy the workflow

Test!!!

```
VARIABLES
  \* database
  log,

  \* frontend
  requests, future,

  \* backend
  backend,

  \* message queue
  queue
```

Consider this to be akin to a transaction; it either happens or it doesn't happen at all. The next state of our system is defined by the primed variables!

```
12FrontendWrite ≜
 \* Precondition: No pending wr
 ∧ future = Nil

 \* Write request to database
 ∧ ∃ o ∈ requests:
    DB!WriteInit(o.key, o.val, LAMBDA t: future' = t)

 \* Nothing else happens.
 ∧ requests'=requests ∧ queue'=queue ∧ backend'=backend
```

(No worries, log' defined inside of WriteInit)

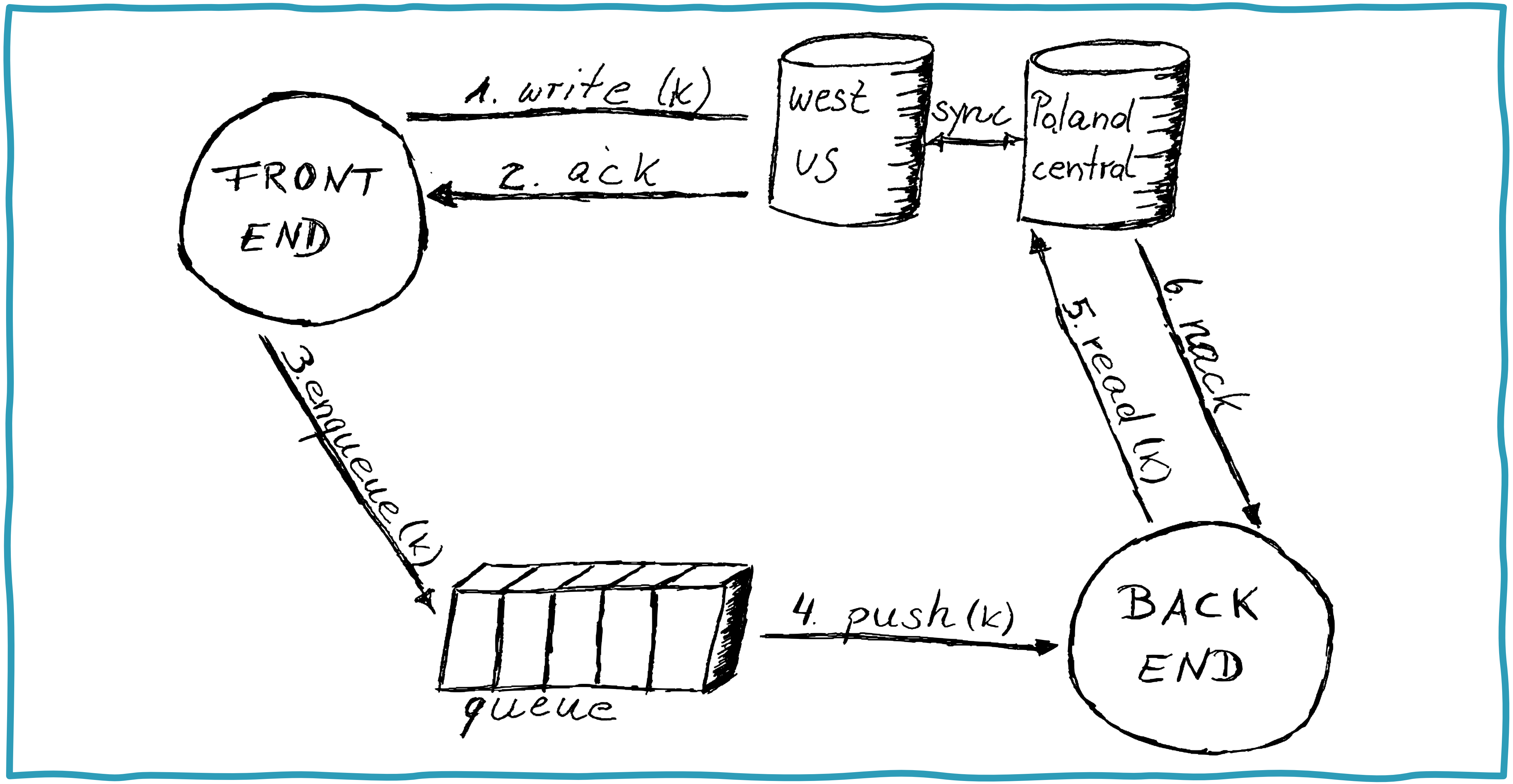
```
34FrontendEnqueue ≜
  \* Precondition: Pending write that succeeded
  ∧ future ≠ Nil ∧ DB!WriteSucceded(future)

  \* Acknowledge write to user
  ∧ requests’ =
     requests \ { [key ↦ future.key, val ↦ future.value] }

  \* Enqueue msg containing key to backend
  ∧ queue' = Append(queue, [k ↦ future.key, t ↦ Nil])

  \* Clear pending write
  ∧ future' = Nil

  \* Nothing else happens.
  ∧ UNCHANGED ⟨log, backend⟩
```

```
56BackendRead ≜
  \* Precondition: Queue contains a msg
  ∧ queue ≠ ⟨⟩

  \* Receive msg from queue
  ∧ LET msg ≜ Head(queue) IN

    \* Read from database (with the given key)
    ∃ o ∈ DB!EventualConsistencyRead(msg.k) :
      backend' = backend ∪ {o.value}

  \* Remove msg from queue
  ∧ queue' = Tail(queue)

  ∧ UNCHANGED ⟨log, requests, future⟩
```

The “Test”

The diamond says that req.val will be **eventually** an element of backend. We don't have to define something like timeouts that leads to slow and brittle tests.

$$BackendReadsCorrectValue \triangleq$$
$$\forall\ req \in (AllRequests \setminus requests):$$
$$\Diamond(req.val \in backend)$$

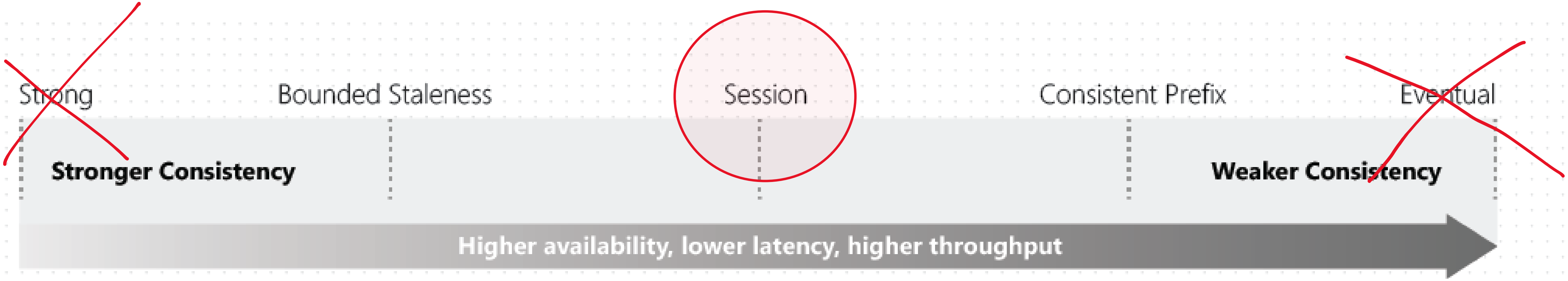


1. write (k)
2. ack
west US
sync
Poland central
FRONT END
3. enqueue (k)
6. nack
5. read (k)
queue
4. push (k)
BACK END

Demo Part II

But at scale!

- More frontend & backend servers
- …
- More queues and databases?
  - Sharding & Partitioning
  - Relax database consistency



Demo C

Pants Down!

Where is the database code?

A database is a just log on steroids!

Database code just 200 LoCs, four variables, and most of the “magic” in the *Read* operator.

GeneralRead(ke
LET maxCandi
∧ log[i].ke
∧ i ≤ index
allIndices ≜
∧ allowDi
∧ log[i].ke
∧ i > index
IN { [logIndex
: i ∈
xCand
THEN {NotF
ELSE {})

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
Microsoft

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 expe

Testing for these scenarios is difficult at 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,

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.

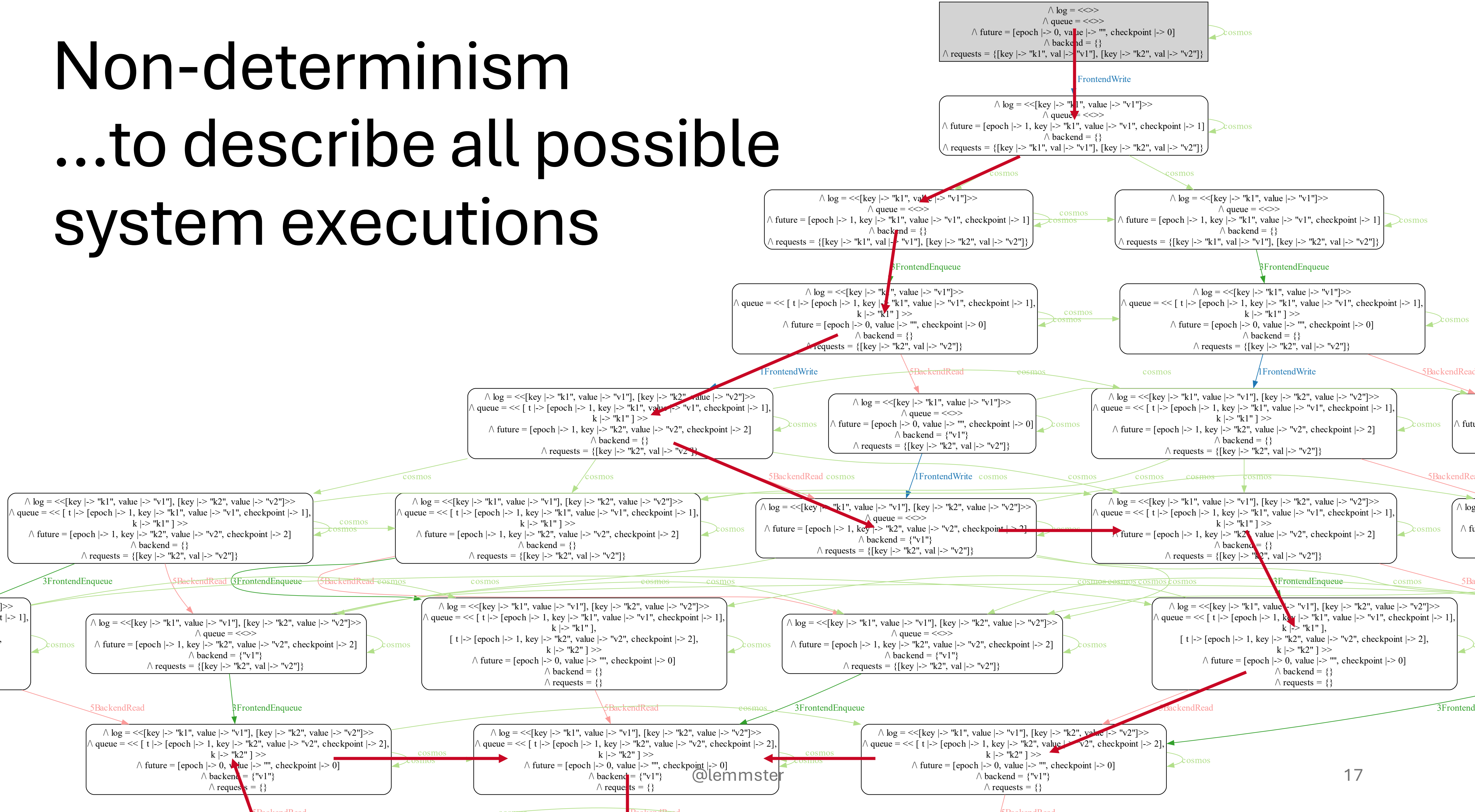
II. Background

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

Non-Determinism

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

Non-determinism
…to describe all possible system executions



Application of TLA+

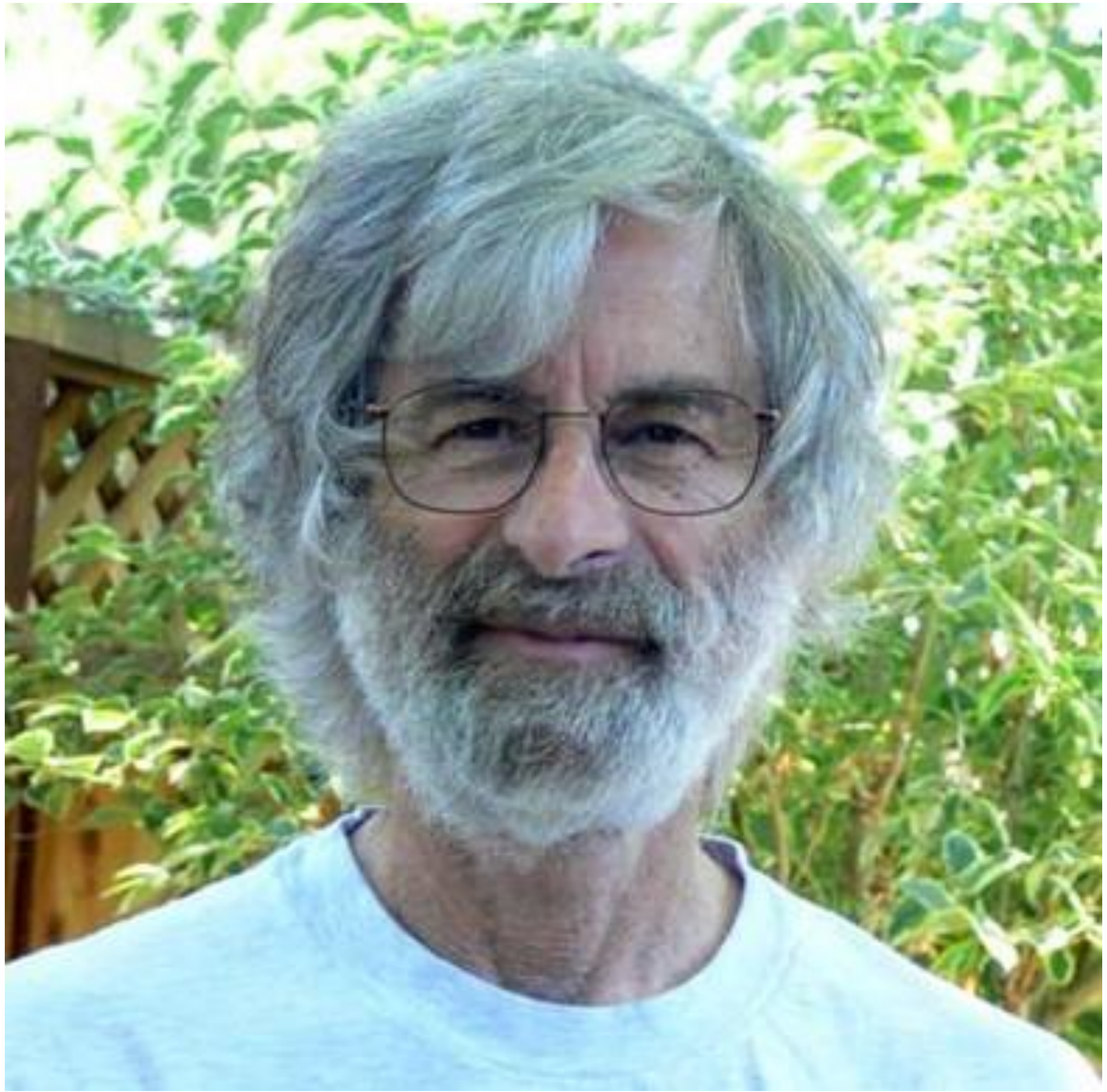
- Consensus (Raft, Paxos, ...)
- Distributed Databases (Cosmos DB, DynamoDB, Mongo DB, ...)
- Hardware (cache coherence, …)
- Network protocols (p2p, …)
- Concurrency primitives (queues, critical section, glibc condition variable, ...)
- Cloud Security (Software Defined *)
- Dogfood
- ...
- Polymerase Chain Reaction (PCR)

Temporal Logic of Actions+

TLA$^+$ is a *specification* language to design, document, and verify reactive systems.

Learn more at http://tlapl.us & http://examples.tlapl.us

TLA+ Webinar June 14th



Question & Answers