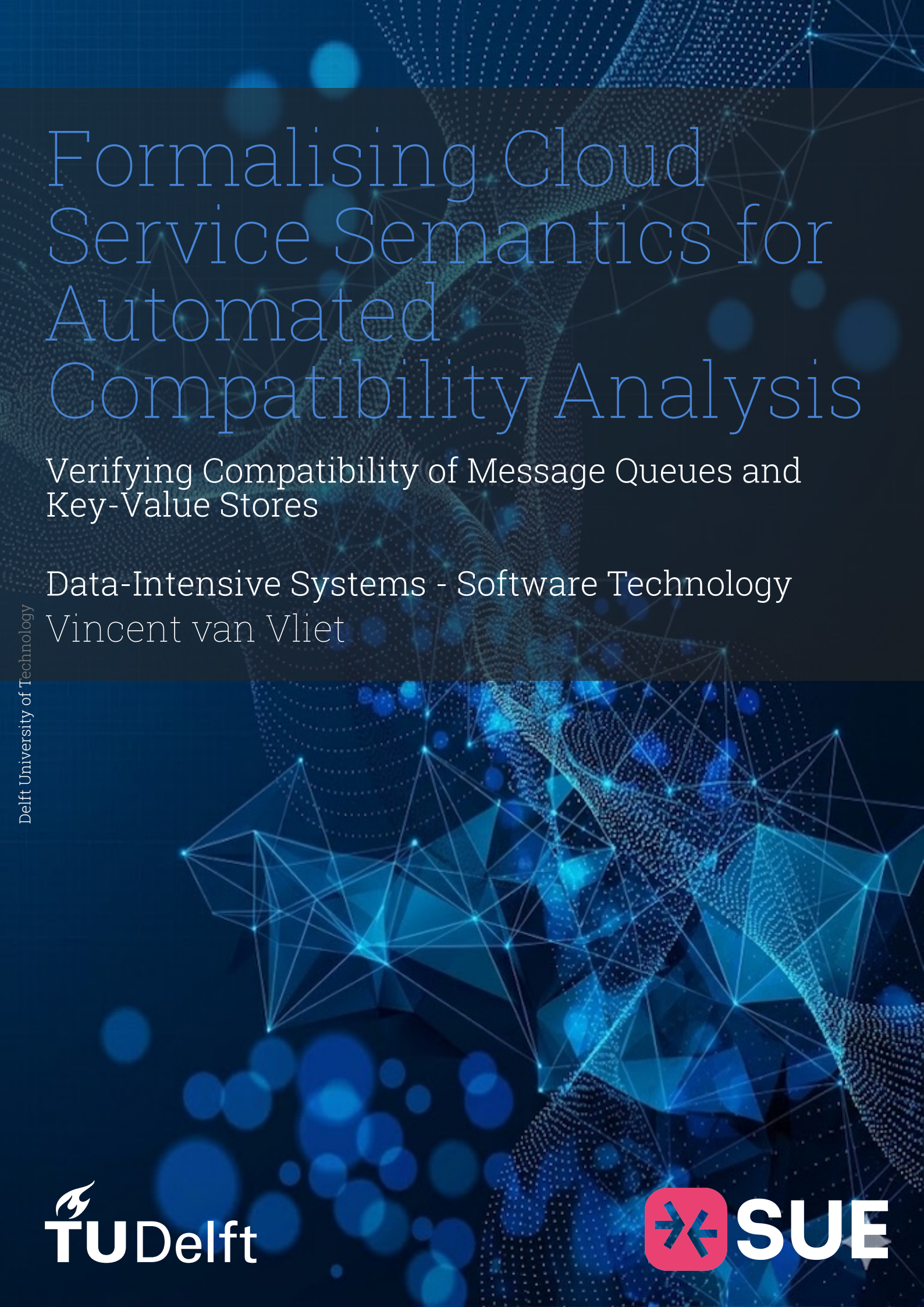
Formalising Cloud Service Semantics for Automated Compatibility Analysis

Verifying Compatibility of Message Queues and Key-Value Stores

Data-Intensive Systems - Software Technology
Vincent van Vliet

Delft University of Technology



TU Delft

SUE

Formalising Cloud Service Semantics for Automated Compatibility Analysis

Verifying Compatibility of Message Queues and Key-Value Stores

by

Vincent van Vliet

Thesis Advisor:	Asterios Katsifodimos, *Delft University of Technology*
External Supervisor:	Reinier Goeman, *SUE Cloud Native*
Project Duration:	November, 2025 - June, 2026
Faculty:	Electrical Engineering, Mathematics & Computer Science, Delft University of Technology

Abstract

While cloud-native architectures promise massive scalability and decoupling, their reliance on managed distributed services introduces a hidden risk: vendor lock-in. Even services with similar functionality and identical APIs may differ in fundamental guarantees such as delivery, ordering, and consistency. These differences could lead to subtle, unpredictable errors during migration. Current infrastructure-as-code tools like Terraform focus on provisioning and fail to capture these underlying behavioural constraints.

To bridge this semantic gap, this thesis introduces a formal framework leveraging the Quint specification language to automate cloud service compatibility analysis. By representing distributed systems as rigorous compositions of partially ordered guarantees, this framework generates specific correctness invariants to verify compatibility between systems, defined in terms of invariant preservation across migrations. The approach is empirically validated through trace-based execution against a concrete Redis instance.

Our compatibility experiments reveal an asymmetry in cloud migrations: while key-value stores exhibit high interoperability (primarily bottlenecked by strict causality), message queue compatibility is highly fragmented by conflicting delivery (`at_most_once`) and ordering (`fifo_ordering`) invariants. Furthermore, we demonstrate an application of this framework by feeding these formal, invariant-based violations back into Large Language Models (LLMs) as structured guidance. Experimental results show this formal feedback drastically constrains the generative search space, reducing the median iterations required to synthesise a correct distributed system by approximately 58%.

Ultimately, this research provides a machine-checkable foundation to safely navigate cloud migrations, moving beyond ad-hoc testing to enable the verifiable design of genuinely cloud-agnostic architectures.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my TU Delft supervisor, Asterios Katsifodimos. Your expertise, guidance, and infectious enthusiasm were invaluable in shaping this thesis and always helped me find my way when I felt stuck. Beyond the academic support, thank you for the wonderful travel recommendations for my upcoming holiday in your homeland, Greece.

I would also like to extend my appreciation to the members of my thesis committee, Burcu Özkan and Jérémie Decouchant. Thank you for generously dedicating your time to reviewing my research and being a part of my defence.

I am deeply grateful to SUE for providing me with the wonderful opportunity to conduct my thesis research within their organisation. A special thank you goes to my external supervisors at SUE: Mike and Reinier. I truly appreciate your day-to-day help and the constant feedback loops that kept this project grounded and improved the quality of the final work.

To my fellow interns, Marc and Camiel, thank you for sharing this journey with me from start to finish. Going through the highs and lows of the thesis process together made the challenges much more manageable, and our games of pool provided the perfect, much-needed breaks from our screens.

Finally, I want to thank my family and my girlfriend. Thank you for your endless patience and support during the long hours of research and writing, and a special thanks for proofreading my entire thesis.

P.S. Thank you, Brandon Sanderson, for providing me with the perfect, fantastical escape from reality into the books of the Cosmere universe. Writing this thesis taught me that the most important step a person can take is always the next one.

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
ALO	At-Least-Once
AMO	At-Most-Once
API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
CRDT	Conflict-free Replicated Data Type
EO	Exactly-Once
FIFO	First-In-First-Out
FSM	Finite State Machine
ITF	Informal Trace Format
JSON	JavaScript Object Notation
KV	Key-Value
LLM	Large Language Model
LWW	Last-Write-Wins
NoSQL	Not Only SQL
P2P	Point-to-Point
Pub/Sub	Publish/Subscribe
RQ	Research Question
SaaS	Software as a Service
SMT	Satisfiability Modulo Theories
SQS	Simple Queue Service
TLA	Temporal Logic of Actions
TLC	TLA+ Model Checker
TOSCA	Topology and Orchestration Specification for Cloud Applications
YAML	YAML Ain't Markup Language

Symbols

Symbol	Definition	Unit
C	Consistency model dimension (Poset)	[-]
D	Delivery semantics dimension (Poset)	[-]
I	Idempotent writes dimension (Poset)	[-]
k	Key-value store guarantee tuple	[-]
M	Messaging model dimension (Poset)	[-]
O	Ordering guarantees dimension (Poset)	[-]
q	Message queue guarantee tuple	[-]
R	Replication dimension (Poset)	[-]
S	Composed system guarantees tuple	[-]
W	Conditional writes dimension (Poset)	[-]
$\leq$	Compatible (Partial order relationship)	[-]

Contents

1

Introduction

The transition from monolithic architectures to cloud-native paradigms has transformed modern software development, with applications increasingly relying on managed, distributed services from cloud providers [1] [2]. Abstractions such as message queues and key-value stores have become the backbone of these systems, enabling decoupling and scalability [3] [4]. However, this dependency often creates a significant business and technical risk known as vendor lock-in [5]. Organisations seeking to migrate between providers or adopt multi-cloud strategies frequently find that moving an application is not merely a matter of changing infrastructure configurations; it involves navigating fundamental dependencies on the specific operational behaviours of the underlying services [6].

Vendor Lock-in

Vendor lock-in is widely recognised as a fundamental challenge in cloud computing, arising when applications become tightly coupled to the services, APIs, and operational semantics of a specific cloud provider [5]. This dependency limits an organisation's ability to migrate workloads, adopt multi-cloud strategies, or switch providers without significant engineering effort and risk. Vendor lock-in is not solely a technical issue, but also a concern that impacts long-term flexibility, cost control, and system evolution [7].

The heterogeneity of cloud platforms intensifies the problem. Even when services appear functionally similar, such as message queues or key-value stores, their underlying guarantees and behaviours may differ significantly. The absence of standardised semantics across providers makes interoperability difficult, and migration non-trivial [8]. Similarly, Amin et al. [9] emphasise that differences often hinder data migration in consistency models, data representations, and system interfaces. These challenges are further expanded by organisational uncertainty towards cloud adoption due to migration complexity [10].

As a result, there is an increasing demand for approaches that reduce reliance on provider-specific implementations. One emerging direction is the design of self-managed systems that replicate the functionality of managed services while offering greater control over system behaviour [11]. However, building such systems requires a precise understanding of the guarantees provided by existing cloud services and the ability to reason about their correctness when re-implemented or substituted.

This thesis addresses this need by proposing a formal framework to capture and compare the semantics of cloud services. By modelling systems in terms of their guarantees and verifying compatibility through formal methods, the approach enables developers to reason about substitutability and migration in a principled manner. In doing so, it helps reduce vendor lock-in by supporting the design and validation of portable, cloud-agnostic system architectures.

1.1. Problem Statement

The core of the vendor lock-in problem in the cloud lies in the semantic gap between seemingly identical services [12]. While two message queue services might share a common API (e.g., enqueue and

deliver), their underlying guarantees, such as message ordering (FIFO vs. best-effort) and delivery semantics (at-least-once vs. exactly-once), can vary wildly. Similarly, key-value stores exhibit diverse consistency models, ranging from eventual to strong consistency, which dictate how and when updates become visible to clients [13].

Current migration tools and standards, such as Terraform [14] and the Topology and Orchestration Specification for Cloud Applications (TOSCA) [15], primarily focus on deployment and resource provisioning. They lack the expressive power to capture the behavioural semantics required for high-assurance compatibility analysis [16]. When a developer migrates an application assuming FIFO ordering to a ”best-effort” target system, the application may fail in subtle, non-deterministic ways that are difficult to detect through traditional empirical testing or benchmarking.

To address these challenges, this thesis proposes using formal methods to bridge the semantic gap, with a focus on cloud message queues and key-value stores. By employing the Quint specification language [17], we can represent cloud services as precise state machines that hold to specific invariants and properties. Formal specifications allow us to move from informal, often ambiguous documentation to machine-checkable models that enable systematic reasoning about system behaviour [18].

However, manually writing formal models for every possible cloud configuration is a time-consuming and error-prone process that requires specialised expertise [19]. To make these methods accessible, this research introduces a generative approach that uses high-level configuration files (YAML) to automatically synthesise formal models and compatibility invariants tailored to specific migration scenarios.

1.2. Research Objectives

The primary goal of this research is to develop a framework that automates the formalisation of the semantics of cloud queues and key-value stores to perform cross-cloud compatibility analysis. Together, the following specific objectives aim to equip software engineers with the tools needed to perform cloud migrations and build truly cloud-agnostic architectures with greater reliability. The objectives are as follows:

- **Formal Specification**: To devise a modular and extensible set of formal building blocks in Quint that capture the core delivery, ordering, and consistency capabilities of cloud systems.
- **Automated Synthesis**: To build a generation engine that can assemble these building blocks into complete system models based on declarative service descriptions.
- **Compatibility Engine**: To develop a methodology for generating specific invariants that identify semantic mismatches when an application moves from a source to a target system.
- **Trace-Based Validation**: To evaluate the fidelity of the formal models by replaying model-generated execution traces against concrete implementations using the Informal Trace Format (ITF).

1.3. Research Questions

To achieve the objectives mentioned in Section 1.2 (Research Objectives), this thesis investigates the use of formal specification and model checking to capture and analyse the semantics of cloud message queues and key-value stores. The goal is to move from informal descriptions to precise, machine-checkable models that enable systematic reasoning about system behaviour.

In this context, the following research questions (RQ’s) are formulated:

- **RQ1: What is a minimal set of guarantees to describe the semantics of cloud queues and key–value stores?**

  Cloud queues and key-value stores expose a range of guarantees, often described using inconsistent or overlapping terminology across providers. Identifying a minimal and expressive set of guarantees for system families would enable a unified representation of their semantics. For this thesis, a set of guarantees is minimal if removing any dimension renders it impossible to distinguish between at least two semantically different real-world systems. This will be evaluated by verifying whether known semantic differences remain distinguishable.

- **RQ2: How can formal specifications be used to compare cloud systems without empirical testing?**

  Traditional approaches to system comparison rely on benchmarking or empirical testing. However, these approaches are time-consuming and may fail to capture corner cases. This question explores whether formal specifications can be used to reason about system behaviour and equivalence, enabling comparison through model checking or simulation rather than execution. Evaluation of this question includes pairwise compatibility analysis across multiple systems.

- **RQ3: To what extent can the guarantees of composed cloud systems be derived from their components?**

  Modern cloud applications are composed of multiple interacting services, such as queues feeding into storage systems. While individual components may provide well-defined guarantees, their composition can introduce unexpected behaviours. This question investigates whether the end-to-end guarantees of a composed system can be systematically derived from the guarantees of its individual components. This will be assessed by checking whether the invariants detect known incompatibilities between system compositions.

- **RQ4: To what extent do formal specifications improve the correctness of LLM-based synthesis of distributed key–value stores?** As Large Language Models (LLMs) increasingly automate the generation of complex software, ensuring the functional correctness of their unverified outputs remains a critical challenge. To test the framework's applicability, this question explores the role of formal specifications as guidance mechanisms in LLM-based system synthesis. Specifically, it investigates whether providing formal system guarantees and invariants improves an LLM's ability to generate correct distributed key-value store implementations.

These research questions collectively aim to improve the understanding of cloud service semantics and interoperability. By providing formal tools to describe, compare, and compose cloud systems, this work seeks to support more reliable cloud migration and the development of cloud-agnostic architectures.

1.4. Thesis Structure

This thesis is structured as follows:

- **Chapter 1: Introduction** This chapter introduces the problem of vendor lock-in in cloud systems, defines the research objectives and questions, and outlines the main contributions of this work.
- **Chapter 2: Background Information** This chapter presents the necessary background on vendor lock-in, cloud service models such as queues and key-value stores, and formal methods. It also introduces invariants, symbolic model checking, and the Quint specification language.
- **Chapter 3: Methodology** This chapter describes the proposed framework for modelling and analysing cloud service semantics. It covers invariant synthesis, automatic model generation, formalisation of system guarantees, and the experimental setup.
- **Chapter 4: Experiments** This chapter details the experimental design used to evaluate the framework. It includes the system compatibility experiment, trace-based evaluation via ITF parsing and execution, and the setup for evaluating LLM-based synthesis.
- **Chapter 5: Results** This chapter presents the results of the experiments, including system compatibility outcomes, trace-based validation results, and observations on the usability of formal specifications in LLM-based system generation.
- **Chapter 6: Discussion** This chapter interprets the results in relation to the research objectives and existing literature. It discusses the implications for cloud-agnostic system design and reflects on the approach's strengths and limitations.
- **Chapter 7: Conclusion** This chapter summarises the main findings of the thesis and reflects on its contributions and impact.
- **Chapter 8: Recommendations and Future Work** This chapter outlines directions for extending the framework, including improving expressiveness, scaling verification techniques, and enhancing integration with automated synthesis methods.

2

Background Information

This chapter establishes the theoretical foundation and context necessary to understand the methodologies and experiments presented in this research. Section 2.1 (Cloud Service Models: Queues and Key–Value Stores) introduces the primary cloud service models examined in this work, key-value stores and message queues, and details their core operational guarantees and properties. Then, Section 2.2 (Formal Methods) provides an overview of formal methods, explaining the critical role of invariants, symbolic model checking, and the Quint specification language. This section also highlights recent advancements in LLM-assisted formal verification, setting the stage for the automated synthesis and validation techniques used in the subsequent chapters.

2.1. Cloud Service Models: Queues and Key–Value Stores

This work focuses on two widely used classes of cloud services: key–value stores and message queues. These abstractions form the backbone of many cloud-native applications and exhibit diverse semantics across providers. Capturing and comparing these semantics is essential for reasoning about cross-cloud compatibility and migration.

2.1.1. Queues

Message queues are asynchronous communication services that enable decoupling between senders (producers) and receivers (consumers). A queue system typically serves as a temporary buffer, allowing producers to store messages and consumers to retrieve and process them at their own pace, with guarantees that depend on the system's semantics.

Justification of Selected Dimensions

The semantic capabilities of a message queue in this thesis are evaluated primarily across three dimensions: delivery semantics, ordering guarantees, and the messaging model. The selection of these specific dimensions is deliberate, as they represent the core operational boundaries that dictate how an application must handle state, concurrency, and error recovery. While this is not an exhaustive list of all possible queue features, omitting non-functional or infrastructural properties such as maximum throughput, latency bounds, message retention limits, or payload size restrictions, these three dimensions constitute an expressive subset. They are the exact functional properties that, if mismatched during a cross-cloud migration, will fundamentally break application correctness.

Delivery Semantics

A key distinguishing feature of queue systems is their delivery semantics, which define how reliably messages are delivered:

- **At-most-once**: Messages may be lost, but are never delivered more than once.
- **At-least-once**: Messages are guaranteed to be delivered, but duplicates may occur.
- **Exactly-once**: Messages are delivered exactly once, without loss or duplication (typically requiring stronger coordination).

These guarantees significantly impact application design, particularly in handling retries and idempotency.

Ordering Guarantees

Queue systems also differ in how they preserve the order of messages, with the following two modes supported within the context of this project:

- **FIFO (First-In-First-Out)**: Messages are delivered in the order they were sent.
- **Best-effort ordering**: Ordering is not strictly guaranteed.

P2P vs Pub/Sub

Two primary messaging models are supported:

- **Point-to-Point (P2P)**: Each message is consumed by a single consumer.
- **Publish/Subscribe (pub/sub)**: Messages are broadcast to multiple subscribers.

These patterns influence how systems scale and how workloads are distributed.

Analysis of Queue System Semantics

The diversity of queue semantics across cloud providers and open-source systems is summarised in Table 2.1. It is important to note that many modern message brokers, such as Apache Kafka, Pulsar, and RabbitMQ, are highly flexible and can be configured to support multiple delivery semantics, ordering guarantees, or messaging models depending on the deployment architecture. However, to maintain a tractable experimental scope, this thesis evaluates each system under a single, representative configuration. The listed properties reflect the most common, default, or structurally defining operational mode for each system used during the compatibility experiments.

System	Delivery Semantics	Ordering	Messaging Model	Source
ActiveMQ	at-least-once	Queue-level	pub/sub	[20]
Apache Kafka	exactly-once	partition-level	pub/sub	[21]
Apache Pulsar	exactly-once	FIFO	pub/sub	[22]
AWS SQS	at-least-once	best-effort	P2P	[23]
AWS SQS FIFO	exactly-once	FIFO	P2P	[23]
RabbitMQ	at-most-once	None	pub/sub	[24]
Redis Queue	at-least-once	FIFO	pub/sub	[25]
RocketMQ	at-least-once	Queue-level	pub/sub	[26]

Table 2.1: Overview of properties of queue systems.

As shown in Table 2.1, systems such as AWS SQS FIFO provide exactly-once delivery and strict FIFO ordering, whereas Standard AWS SQS defaults to at-least-once delivery and best-effort ordering. This discrepancy represents a significant "Semantic Gap." When migrating an application from a FIFO-based source to a best-effort target, the compatibility engine identifies a mismatch in the partial order:

$$SQS \not\geq SQS\_FIFO$$

This is due to stricter constraints in *SQS_FIFO*, meaning *SQS* would fail to uphold the corresponding `exactly_once` and `FIFO` invariants. The formalised lattice used to define compatibility between systems will be further defined in Section 3.3 (Formalization of System Guarantees). In a practical migration, the subtle shift to at-least-once delivery might be overlooked. However, the framework detects this through the `exactly_once` invariant. In this case, the system could generate a counterexample in which a message is delivered multiple times, violating the requirement that each unique message ID in history corresponds to exactly one event.

2.1.2. Key-Value Stores

Key-value stores are storage systems that maintain a mapping from keys to values. They are widely used for scalable storage due to their simplicity and efficiency. Formally, a key–value store can be modelled as a simple function:

$$KV : K \to V$$

where K is the set of keys and V is the set of values. The system state corresponds to the current mapping, and operations such as `put`, `get`, and `delete` define transitions over this state.

In distributed settings, key–value stores introduce additional complexity due to replication, consistency, and concurrency.

Core Properties

Key–value stores differ along several important dimensions:

- **Consistency model**: Defines how updates become visible (e.g., strong, eventual, tunable).
- **Conditional writes**: Support for atomic updates based on predicates (e.g., compare-and-set).
- **Idempotent writes**: Whether repeated writes produce the same effect without unintended side effects.
- **Replication and backups**: Mechanisms for fault tolerance and availability.
- **Conflict resolution**: Strategies for handling concurrent updates (e.g., last-write-wins, vector clocks).

These properties directly affect correctness when migrating applications between systems.

Justification of Selected Dimensions

The dimensions of consistency, conditional writes, idempotence, replication, and conflict resolution were chosen because they define the strictness of state transitions and data visibility in a distributed environment. Similar to the queue models, this does not represent a complete, exhaustive feature list. Factors such as partition-tolerance metrics, specific hashing algorithms, storage tiering, and geographical distribution are excluded from the formalisation. However, this focused set of dimensions provides a highly workable abstraction. It captures the essential behavioural constraints that applications rely upon to maintain data integrity, ensuring that our compatibility analysis focuses on the semantic guarantees that prevent data corruption and visibility gaps during migration.

It is important to note that while replication is included as a binary capability dimension in this framework, the internal mechanics of replica consistency, such as specific read/write quorum configurations, gossip protocols, or leader-election algorithms, are deliberately abstracted away. Modelling these internal mechanics in depth would require exhaustive definitions of replica-consistency levels and would drastically expand the state space. Instead, this framework evaluates replication strictly from the perspective of the application layer: whether the system provides baseline data redundancy and whether the resulting consistency model (strong vs. eventual) preserves the application's required data visibility invariants.

Analysis of Key-Value Store Semantics

Table 2.2 summarises the key properties of several widely used key-value stores, highlighting the fundamental trade-offs inherent in distributed storage systems [27]. Similar to the queue systems, distributed databases such as Cassandra, CosmosDB, and Redis offer highly tunable consistency models and configurable conflict resolution strategies (e.g., allowing developers to switch between eventual and strong consistency). For this research and the subsequent formal verification experiments, the analysis is deliberately constrained to a specific, widely adopted configuration for each system. This targeted selection ensures a focused mathematical evaluation of distinct semantic combinations without overcomplicating the formal models with every possible deployment variation.

System	Consistency	Conditional Writes	Idempotent Writes	Backups	Source
Cassandra	Strong	No	Yes	Multi-node replication	[28]
CosmosDB	Strong	Yes	Yes	Multi-region replication	[29]
DynamoDB	Eventual	Yes	Yes	Multi-AZ replication	[30]
Memcached	None (cache)	No	Yes	No	[31]
MongoDB	Eventual	Yes	Yes	Replica sets + sharding	[32]
Redis	Eventual	No	Yes	Primary-replica + persistence	[33]

Table 2.2: Overview of properties of key-value store systems.

For instance, migrating from Redis (Single Node) to DynamoDB (Eventual Consistency) introduces a visibility gap. While Redis ensures that updates are immediately visible globally due to its single-node architecture, DynamoDB's default eventual consistency can lead to stale reads. Our framework formalises this as an incompatibility in the consistency lattice:

$$EVENTUAL \leq_C STRONG$$

This mismatch is identified via the `consistency_with_history` invariant. In the formal model, a write transition in the target system may update only a subset of replicas, allowing a subsequent read transition to return an older value. This violation is critical for applications assuming strong consistency for state synchronisation or session management.

Simplifying Conflict Resolution with LWW

While distributed key-value stores use a wide variety of conflict resolution strategies, ranging from vector clocks [34] and CRDTs [35] to complex, application-defined logic, modelling these diverse mechanisms introduces unwanted state-space complexity during formal verification. To reduce this computational overhead and keep the formal models tractable, conflict resolution in this framework is uniformly implemented using Last-Write-Wins (LWW). LWW deterministically resolves concurrent updates by prioritising the write operation with the highest logical timestamp. This design choice provides a straightforward, industry-standard baseline (commonly used by systems such as Cassandra and DynamoDB) that enables the framework to evaluate eventual consistency and state convergence without the heavy burden of tracking complex, multi-versioned conflict histories.

2.1.3. Relevance to this work

Both queues and key–value stores exhibit semantic diversity between implementations, despite offering similar high-level functionality. This diversity comes from differences in delivery guarantees, ordering, consistency, and replication strategies.

In this thesis, these systems are modelled using Quint to:

- Formally capture their operational semantics,
- Define properties and guarantees,
- Analyse compatibility between systems.

By establishing these distinct operational dimensions, this chapter directly addresses RQ1 by defining a minimal, expressive set of guarantees that describes the semantics of cloud queues and key-value stores. Furthermore, laying this theoretical groundwork paves the way for answering RQ2. By expressing these systems within a unified formal framework, it becomes possible to systematically evaluate whether one system can safely replace another in a migration scenario, or whether semantic mismatches may lead to incorrect behaviour.

2.2. Formal Methods

The migration of safety-critical or highly available applications between cloud environments requires absolute certainty regarding system behaviour. However, cloud service providers typically document their operational guarantees, such as delivery semantics, ordering, and data consistency, using natural language. These informal descriptions are inherently ambiguous, frequently masking subtle implementation details, and lack the mathematical precision necessary to guarantee safe substitutability across platforms.

To address this semantic gap, this research employs formal methods. By abstracting distributed systems into mathematically rigorous models, such as finite-state machines, and evaluating them using temporal logic, it becomes possible to systematically capture and analyse the nuanced operational differences between seemingly identical cloud services. This section introduces the core formal methodologies utilised in this framework to specify, simulate, and mathematically verify the behaviours of cloud message queues and key-value stores.

2.2.1. Invariants

To effectively evaluate the interoperability of modern cloud services, informal descriptions are insufficient, particularly for safety-critical migrations. Instead, system behaviours and subtle semantic differences can be formally captured using Finite State Machines (FSM) and temporal logic. Within this formal framework, the fundamental guarantees of a system are typically expressed as invariants.

An invariant is a precise condition or property that must hold in every reachable state of the system throughout its execution. In the context of distributed systems, maintaining these invariants is notoriously challenging due to non-determinism, network asynchrony, and complex operation interleavings [36]. For instance, specific schedules of concurrent operations under relaxed consistency models can easily lead to subtle invariant violations that are nearly impossible to reproduce through empirical testing alone [37].

To bridge the gap between theoretical models and practical system evaluation, recent advancements in formal methods emphasise the combination of rigorous invariants with targeted exploration. Research illustrates how formal execution traces and probabilistic guarantees can be utilised to guide the discovery of deep concurrency bugs in distributed protocols [38] [39]. In this thesis, formally defining invariants serves a similar foundational purpose: they establish the absolute boundaries of correct system behaviour, providing the necessary mathematical constraints to verify the absence of data loss, the preservation of ordering, and the correctness of state updates across cloud service replicas.

2.2.2. Symbolic Model Checking

While defining invariants establishes the theoretical correctness criteria for a system, verifying that these properties hold requires rigorous automated analysis. Symbolic model checking is a highly effective technique for this purpose, specifically tailored to evaluate systems defined by Temporal Logic of Actions (TLA)+ semantics. Unlike explicit-state model checking, which must enumerate and visit every individual system state, symbolic model checking represents states and state transitions logically.

To address this, symbolic techniques represent states and state transitions logically rather than explicitly. Symbolic constraints can be utilised to evaluate and verify programs operating under diverse consistency models without exhaustively enumerating every possible state variation [40].

Building on these symbolic principles, this framework leverages tools such as Apalache, a symbolic model checker explicitly designed for specifications written in TLA-like languages [41]. Apalache verifies system properties by translating the formal specification into logical constraints, specifically Satisfiability Modulo Theories (SMT). These resulting SMT constraints are subsequently evaluated and solved using underlying SMT solvers, such as Z3 [42]. By resolving these logical formulas, symbolic model checking enables highly efficient reasoning about complex system behaviours and concurrency within bounded executions, inherently supporting the automated verification of the defined invariants even under the varying operational semantics of different cloud providers.

2.2.3. LLM-Assisted Formal Verification and Specification Synthesis

While formal methods provide robust mathematical guarantees, a critical bottleneck to scaling these techniques is the heavy human burden of writing formal specifications [43]. To address this challenge, recent research has increasingly turned to Large Language Models (LLMs) to assist in both specification synthesis and automated proof generation [44, 45].

The process of translating informal requirements into formal specifications is known as user-intent formalisation [46]. Evaluating the quality of these generated specifications mechanically is inherently difficult because informal intent is typically expressed in natural language, while the resulting specification is a strict formal artifact. To automate this evaluation without relying on human review, recent frameworks propose symbolically testing the specifications against a set of input-output examples [46]. Specifically, a specification's quality can be measured through two metrics: correctness, which ensures the specification is consistent with all valid test cases, and completeness, which measures its ability to reject mutated or incorrect output values [46].

Building upon these evaluation metrics, advanced automated frameworks have been developed to generate specifications and formal proofs without relying on massive, human-annotated datasets. For example, the SAFE framework utilises a self-evolving cycle in which LLMs generate specifications,

and a symbolic verifier evaluates their correctness and completeness scores to filter out low-quality outputs [44]. The surviving high-quality specifications are then used to fine-tune the models, creating a feedback loop that continually improves the model's capabilities in generating formal proofs [44].

Similarly, the AUTOVERUS framework demonstrates that LLMs can automatically generate correctness proofs by mimicking human experts' workflows [45]. By orchestrating a network of specialised agents through phases of preliminary generation, refinement, and verifier-guided debugging, LLMs can iteratively repair complex proofs based on specific error messages returned by the underlying formal verifier [45].

To scale these automated reasoning techniques to massive, real-world systems, the generation of specifications must accurately capture the developer's architectural intent rather than simply mirroring potentially buggy implementations. The FM-AGENT framework addresses this challenge by employing a top-down paradigm in which the specification of a callee function is dynamically inferred from how its callers expect it to behave [43]. Furthermore, by generalising Hoare-style inference to operate directly over natural-language specifications, such frameworks enable compositional reasoning across large codebases without strictly requiring developers to write mathematical formulas for every component [43].

These advancements show a fundamental synergy: LLMs possess the generative creativity to produce complex code and annotations, while formal methods provide the definitive, strict feedback required to verify their correctness. This powerful intersection forms the theoretical basis for the experimental approach in Chapter 4.4 (Evaluating the Impact of Formal Specifications on LLM-Based Synthesis of Distributed Systems), where formal specifications and invariant-based feedback are utilised as an automated correctness guide for an LLM iteratively synthesising a distributed system.

2.2.4. Quint Specification Language

To formally model and analyse the semantics of cloud systems, this work uses Quint specification language [17], an executable specification language designed to describe and verify distributed systems. Quint was chosen because it specialises in the formal methods required to build specifications of the given systems. Quint enables developers and researchers to describe system behaviour at a high level of abstraction while providing automated tooling for simulation and formal verification.

Quint builds on ideas from TLA [47], a formal framework widely used to specify concurrent and distributed systems. Similar to TLA-based approaches, Quint models systems as state machines, in which system behaviour is defined in terms of states and transitions between them. However, Quint introduces a modern syntax, strong typing, and improved tooling that make it more accessible for practical use in software engineering workflows.

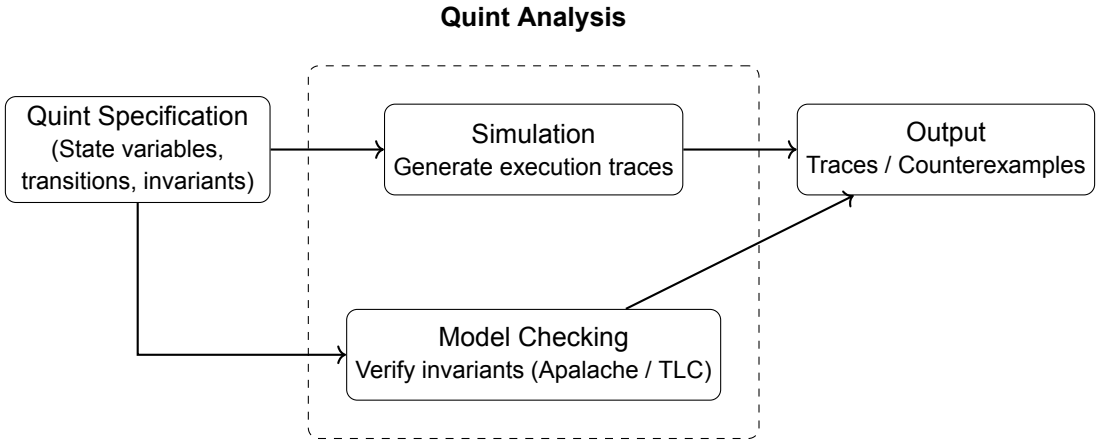


Figure 2.1: Workflow of the Quint specification and verification process. A system is specified as a transition system with invariants, after which simulation and model checking are used to explore behaviour and verify correctness properties.

Quint is used to describe models and properties of systems [17]. A system model is typically expressed as a transition system, where states represent possible configurations of the system and transitions describe how the system evolves between states. Once a model has been defined, verification tools can analyse the system to determine whether specified properties hold for all possible executions. Before moving on to an explanation of the pipeline steps, an overview of the Quint analysis pipeline is shown in Figure 2.1.

System Modelling

In Quint, a system is described as a transition system. The state of the system is represented by a set of state variables, each capturing part of the system's configuration at a point in time. With each step, the state variables are updated to reflect the current state of the system. The evolution of the system is defined by actions that specify how the state variables can change from one state to the next.

In Quint, system behaviour is described by defining:

- **State variables**, which represent the system state (e.g., a `queue` mapping to hold pending messages, or a `history` variable tracking all processed operations).
- **Initial conditions**, defining valid starting states (e.g., initialising all replica data stores to an empty state, or setting the `nextMessageId` to 1).
- **Transition relations**, specifying how state variables may evolve (e.g., an `enqueue` action that appends a message object to a list, or a `sync` action that propagates key-value pairs between distributed replicas).
- **Properties**, typically expressed as invariants that must hold for all reachable states (e.g., a `fifo_ordering` invariant verifying that messages are delivered in exact sequence, or a `no_double_processing` invariant ensuring exactly-once execution).

Transitions are expressed through next-state relations, where the value of a variable in the next state is denoted with a prime symbol. For example, transitions between states are commonly expressed using next-state expressions such as $x' = x + 1$, which indicates that the value of x in the next state is derived from the current state.

This representation makes Quint particularly well-suited to modelling distributed systems and cloud services, where behaviour is naturally expressed as a sequence of state transitions (e.g., enqueuing messages, delivering messages, or updating key–value entries).

Simulation and Verification

Quint specifications can be analysed using both simulation and model checking. Using a built-in simulator, Quint executes the specification to produce concrete traces of system behaviour. During simulation, the tool explores possible transitions starting from the initial state and generates execution traces that represent potential system runs. Randomised simulation allows developers to explore a variety of behaviours and can help detect specification errors or unexpected interactions early in the modelling process. This process explores possible execution paths and helps identify potential property violations through randomised exploration of the state space.

For stronger guarantees, Quint integrates with model checkers to exhaustively explore the system's reachable state space. During model checking, the model checker evaluates whether specified properties hold for all reachable states. If a property is violated, the model checker produces a counterexample trace, showing a concrete sequence of state transitions that leads to the violation. Such traces are useful for diagnosing errors in system designs or specifications. In this research, the traces can be used to replay exact transitions on separate systems, as shown in Section 4.3 (Trace-Based Evaluation via ITF Parsing and Execution).

Properties are commonly expressed as invariants, which must hold in every reachable state of the system. Invariants are particularly useful for verifying safety guarantees such as the absence of data loss, the preservation of ordering constraints, or the correctness of state updates.

Verification Backends

Quint integrates with existing model checking tools to perform verification. In particular, it can leverage Apalache [41], a symbolic model checker designed for specifications written in TLA-like languages.

Apalache verifies properties by translating the specification into logical constraints in Satisfiability Modulo Theories (SMT), which are solved using SMT solvers such as Z3 [42]. This approach enables efficient reasoning about system behaviour within bounded executions and supports automated invariant checking.

Additionally, Quint specifications can be analysed using explicit-state model checking using the TLC model checker for TLA+ [48]. To utilise this model checker, Quint can transpile specifications into TLA+. TLC explores the full reachable state space through enumeration and is particularly effective when system domains are finite and the state space remains manageable.

Tooling and Workflow

The Quint toolchain provides a command-line interface for developing and analysing specifications. Typical workflows involve writing a specification, performing type checking to ensure correctness, exploring system behaviour through simulation, and verifying system properties using a model checker.

This iterative workflow allows specifications to evolve alongside system understanding. Early simulation helps validate modelling decisions, while formal verification provides stronger guarantees about system correctness once the model stabilises.

Relevance to this work

In this thesis, Quint is used to formally specify the semantics of cloud services such as message queues and key–value stores. Each system is modelled as a transition system that describes the behaviour of operations (e.g., enqueue, dequeue, read, or write) and the evolution of the system state. By expressing the behaviour of different cloud services within the same formal framework, it becomes possible to reason about their semantics and compare their guarantees.

The ability to formally define system semantics and verify properties using model checking makes Quint particularly suitable for analysing compatibility between different cloud services. While TLA+ could be used to achieve the same goal, this project focuses on Quint for its ease of use, modern syntax and capabilities to handle all project requirements. By modelling services from different providers within a unified formal framework, it becomes possible to evaluate whether their behaviours are semantically equivalent or whether certain migration scenarios may lead to violations of expected guarantees, which forms the core objective of this research.

While alternative formal verification frameworks exist, such as Coq and Alloy, Quint was selected as the most appropriate tool for this research. Coq provides an exceptionally powerful interactive theorem prover, but it requires extensive manual proof construction and a steep learning curve, making it less practical for the highly automated, generative synthesis pipeline proposed in this framework. Alloy is highly effective for structural modelling and finite-bound checking using relational logic, but it is less naturally suited for defining and simulating the temporal, transition-based state machines characteristic of distributed cloud services.

As previously mentioned, TLA+ could theoretically be used to achieve the same goal. However, this project focuses on Quint for its ease of use, its modern developer-friendly syntax, and its robust built-in capabilities. By natively supporting both randomised trace simulation and seamless integration with the Apalache symbolic model checker, Quint provides the exact tooling required to model heterogeneous cloud providers, generate concrete execution traces, and systematically evaluate semantic compatibility.

3

Methodology

This chapter presents the methodology used to formalise, synthesise, and evaluate chosen cloud service semantics for automated compatibility analysis. The approach is designed to bridge the gap between high-level service descriptions and formal verification by combining specification synthesis, invariant generation, and model checking into a unified pipeline.

At a high level, the methodology treats cloud services not as concrete systems, but as compositions of well-defined semantic guarantees, such as delivery semantics, ordering constraints, and consistency models. These guarantees are specified declaratively using configuration files, which serve as input to a generation pipeline that produces formal models and associated correctness properties. By operating on structured representations of system capabilities, the approach enables systematic reasoning about compatibility without requiring direct interaction with the underlying implementations.

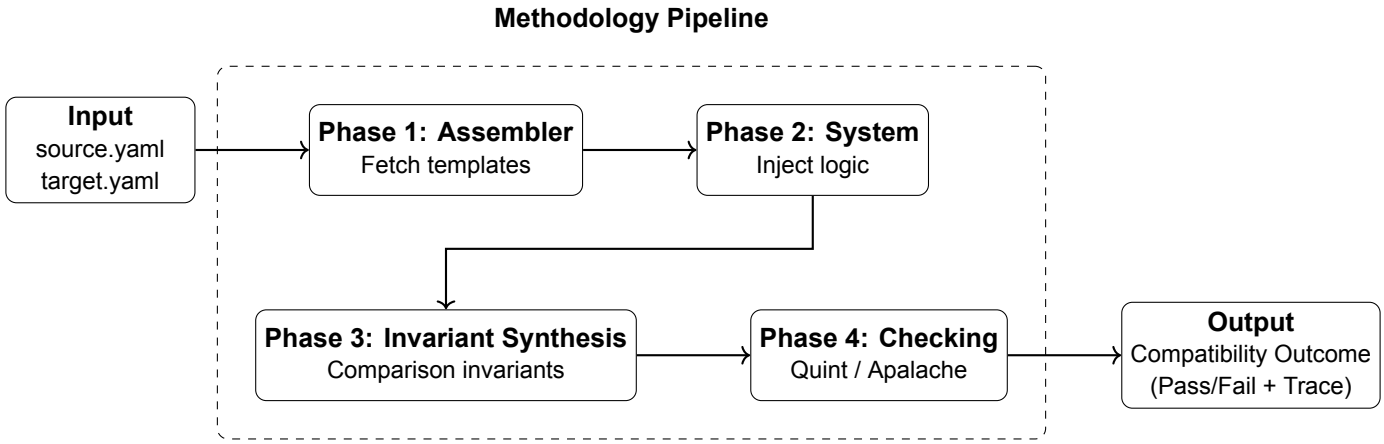


Figure 3.1: Methodology pipeline for automated compatibility analysis.

The overall workflow is illustrated in Figure 3.1. The process begins with user-provided configuration files describing the source and target systems. These configurations are used to assemble base model templates, which are subsequently extended by injecting logic corresponding to selected guarantees. From these synthesised models, a set of comparison invariants is generated to capture the correctness properties that must be preserved during migration. Finally, the resulting specification is verified using model checking, producing a compatibility report indicating whether the target system satisfies the source system's guarantees, along with counterexample traces in the event of violations.

This structured pipeline enables automated, repeatable, and formally grounded compatibility analysis. The following sections describe the main components of the methodology in detail, including the synthesis process, invariant generation, and the verification workflow.

3.1. System Generation

This section describes the process by which the formal system specifications are automatically generated from high-level configuration files. The goal of this generation step is to bridge the gap between declarative descriptions of cloud services and executable formal models that can be analysed using model checking.

Rather than manually writing formal specifications for each system, this work adopts a generative approach in which systems are constructed from reusable building blocks. Each system is defined as a composition of guarantees, specified in a structured configuration format.

The generation process transforms these configurations into complete Quint specifications through a multi-stage pipeline, as illustrated in Figure 3.1. This enables systematic and scalable exploration of different system configurations without requiring manual modelling effort.

The input to the system generation pipeline consists of YAML configuration files, a human-readable data serialisation language commonly used for configuration [49]. These files describe the semantic properties of a system along predefined dimensions. The exact properties that can be added to the configuration are mentioned in Section 2.1 (Cloud Service Models: Queues and Key–Value Stores).

3.1.1. Template-based Assembly

The generation process begins with selecting base templates that define the system's core structure in Quint. These templates include:

- State variables (e.g., message queues, key–value mappings, history tracking)
- Initial conditions and values
- Core transition relations (e.g., enqueue, dequeue, put, get, sync)

At this stage, the templates describe only the system's basic operational behaviour, without enforcing any specific guarantees. This corresponds to a minimal baseline model that is intentionally permissive.

The choice of template-based assembly, rather than direct transpilation from YAML to Quint, was made to balance extensibility and maintainability. A direct compiler approach would require maintaining a complex, custom mapping of YAML syntax to Quint's internal grammar, which is brittle and difficult to debug. Templates, conversely, provide human-readable, pre-verified building blocks that guarantee syntactic correctness out of the box. By injecting logic into predefined hooks within these templates, the framework ensures the generation of valid specifications while keeping the underlying generation logic transparent and accessible for future extensions.

3.1.2. Injection of System-Specific Logic

In the second phase, the generator augments the base templates by injecting logic corresponding to the guarantees of the selected systems. This is implemented through a set of modular code fragments, each associated with a specific guarantee. For example:

- Delivery semantics influence how messages are removed or retained after processing
- Ordering guarantees constrain the selection of messages for delivery
- Consistency models determine how and when updates propagate across replicas
- Conditional writes introduce additional guards on state transitions

These fragments modify the transition relations and, in some cases, extend the system state (e.g., by adding metadata such as message identifiers or version histories). The injection process is compositional, as multiple guarantees can be combined by merging their corresponding logic into a single specification. This enables the construction of a wide range of system variants from a shared set of building blocks.

3.1.3. System Composition

For experiments involving multiple components (e.g., a queue feeding into a key–value store), the generator composes individual system models into a single unified specification ($KVQueue$). This

composition involves:

- Combining state variables from both subsystems
- Linking transitions (e.g., a dequeue operation triggering a write)
- Sharing auxiliary structures such as the history variable

The resulting model captures the end-to-end behaviour of the composed system, enabling analysis of interactions between components. The final output of the generation pipeline is a complete Quint specification (.qnt file) containing fully defined state variables and transitions, injected guarantee-specific behaviour and hooks for invariant checking (see Section 3.2 (Invariant Synthesis)). This specification can be executed directly using the Quint toolchain for simulation and model checking, and an example of a possible Quint output is provided in Appendix A (Generated System Code Example).

While this compositional approach effectively models the data flow between components, it is important to acknowledge its inherent limitations. First, combining separate state machines into a single unified specification increases the state space in model checking, making exhaustive verification significantly more computationally expensive. Furthermore, this method abstracts away the network layer between the components, as potential real-world failures are not natively captured by simply merging state variables. Consequently, the composed model assumes a highly reliable interconnect, focusing strictly on the semantic interactions between the endpoint services rather than the volatility of the network linking them. These limitations will be further expanded on in Section 6.5 (Limitations).

3.2. Invariant Synthesis

This section describes how correctness invariants are automatically generated based on the guarantees of the synthesised systems. While Section 3.1 (System Generation) focuses on constructing executable system models, this section introduces the mechanism used to derive the properties that these systems must satisfy.

In distributed systems, correctness is typically defined in terms of invariants: properties that must hold in every reachable state of the system. Rather than manually specifying such invariants for each system configuration, this work adopts a generative approach in which invariants are derived automatically from the declared system guarantees.

The key idea is that each guarantee implies a set of correctness properties. By encoding these relationships explicitly, invariants can be synthesised systematically and at scale.

3.2.1. Invariant Templates

The invariant synthesis process is based on a collection of reusable invariant templates. Each template captures a general correctness property, parametrised over system state variables such as message histories, key-value mappings, or replica states.

Examples of such templates include:

- **FIFO ordering:** Messages must be processed in the same order as they were enqueued.
- **Consistency with history:** The current state of the key–value store must reflect a valid projection of the sequence of write operations.
- **Exactly-once processing:** Every processed message must correspond to a unique message identifier and may not be processed more than once.

Each template is expressed as a Quint invariant and can be instantiated based on the structure of the generated system.

3.2.2. Capability-Driven Selection

Not all invariants apply to all systems. In the context of distributed systems, applying strict safety invariants (such as linearizability or strict ordering) to a system designed for high availability and weak consistency will result in immediate, expected verification failures. These ”false alarms” pollute the verification process and obscure genuine architectural flaws. Therefore, the inclusion of a particular invariant strictly depends on the guarantees specified in the input configuration.

To support this dynamic assembly, each invariant template is associated with a set of preconditions over system capabilities. Formally, let G denote the set of operational guarantees explicitly selected for a system configuration. An invariant I from the global pool of potential invariants $\mathcal{I}$ is included in the generated specification $\mathcal{I}_{generated}$ if and only if its associated preconditions are fully satisfied by the system's guarantees:

$$I \in \mathcal{I}_{\text{generated}} \iff \text{preconditions}(I) \subseteq G$$

By treating the invariant selection as a logical mapping of capabilities, the framework ensures that it checks only for properties the system claims to support. For example:

- The `fifo_ordering` invariant is included only if ordering is specified.
- Consistency invariants depend on the chosen consistency model (e.g., strong vs. eventual).
- The `exactly_once` invariant is included only if the queue delivery guarantee is `ALO` and KV has `strong` consistency. This is an example of an invariant that verifies the guarantees of system composition rather than just the queue's delivery guarantees. The interaction between the system guarantees leads to different expected behaviour.

This conditional inclusion ensures that invariants are both sound and relevant, reflecting actual guarantees while avoiding over-constraining weaker systems.

Ultimately, this conditional inclusion mechanism acts as a precise semantic filter. It ensures that the generated invariants are both sound and tailored to the actual semantics of the modelled cloud systems.

3.2.3. Invariant Composition

A central component in invariant synthesis is the `history` variable, which records the logical sequence of operations performed by the system (e.g., message deliveries or write operations). The history serves as a canonical reference against which correctness properties can be defined. For example:

- Delivery invariants relate processed messages to entries in the history.
- Ordering invariants enforce constraints over the ordering of events in the history.
- Consistency invariants ensure that the observable state is derived from the history in a valid manner.

By expressing invariants in terms of history rather than implementation-specific state, the approach achieves a higher level of abstraction and generality. For composed systems (e.g., a queue connected to a key-value store), invariants are generated across component boundaries. These invariants capture end-to-end correctness properties, such as: messages written to the key-value store must originate from previously processed queue messages or each processed message must result in a corresponding state update.

Such invariants are derived by combining guarantees from multiple subsystems, enabling reasoning about interactions and data flow across components.

The output of the invariant synthesis phase is a set of Quint invariants that are appended to the generated system specification. An example of a possible set of generated invariants can be found in Appendix B (Generated Invariants Code Example). These invariants are then used during simulation and model checking to verify correctness. If an invariant is violated, the model checker produces a counterexample trace, which can be used to identify the precise conditions under which the system fails to satisfy its guarantees. A full list of invariants can be found in Appendix D (List of Invariants).

3.3. Formalization of System Guarantees

To reason about compatibility between distributed systems, we formalise system guarantees as elements of a structured partially ordered space. This enables us to define when one system can safely substitute another based on their respective guarantees.

We model system guarantees along multiple independent dimensions. For queue systems, we consider delivery semantics, ordering guarantees, and the messaging model. For key-value stores, we consider consistency, conditional writes, idempotence, and replication.

Each dimension is represented as a partially ordered set (poset), capturing the relative strength of guarantees.

3.3.1. Queue Guarantees

Queue systems are formally characterised along three semantic dimensions: delivery semantics, ordering guarantees, and the messaging model. By representing each of these dimensions as a partially ordered set (poset), we can mathematically evaluate whether a target system provides guarantees that are at least as strong as those required by the source system.

Delivery Semantics

Delivery semantics represent the reliability of message transmission between producers and consumers. Within our formalisation, these semantics form a partially ordered set (poset) reflecting the strictness of guarantees, as shown in Figure 3.2.

- **At-most-once (AMO)**: Prioritises throughput over reliability; messages may be lost but are never duplicated (delivery count is ≤ 1).
- **At-least-once (ALO)**: Ensures no messages are lost, though network retries may result in duplicate deliveries (delivery count is ≥ 1).
- **Exactly-once (EO)**: Represents the strongest guarantee: messages are delivered once and only once (delivery count is $= 1$), requiring robust internal coordination to prevent both loss and duplication.

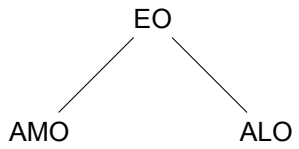


Figure 3.2: Partially ordered set (poset) of delivery guarantees.

Crucially, `AMO` and `ALO` are incomparable. An application built for `AMO` relies on the guarantee that duplicates will never occur and may lack idempotency. Migrating this application to an `ALO` system introduces duplicates, fatally violating its invariants. Conversely, migrating an `ALO` application to an `AMO` system violates the assumption of guaranteed delivery.

However, `EO` represents the strongest guarantee: messages are delivered once and only once, requiring robust internal coordination to prevent both loss and duplication, and safely subsumes both. Delivering a message exactly once perfectly satisfies both `AMO`'s constraint (by not duplicating) and `ALO`'s constraint (by not dropping). Formally, we define this as:

> Let $D = \{\text{AMO}, \text{ALO}, \text{EO}\}$, representing at-most-once, at-least-once, and exactly-once delivery, respectively. We define a total order:

$$AMO \leq_D EO \quad \wedge \quad ALO \leq_D EO$$
$$AMO \not\leq_D ALO \quad \wedge \quad ALO \not\leq_D AMO$$

This reflects the partially ordered set of delivery guarantees.

Ordering Guarantees

Ordering guarantees define the constraints on the sequence in which messages are delivered relative to their enqueued order. In distributed messaging, ordering is traditionally categorised by granularity, such as global, queue, or partition level.

However, because the queue models defined in this framework do not implement partitioning or sharding mechanics, these granular distinctions effectively collapse into a single strict ordering property. Consequently, in this formalisation, ordering is simplified to a binary state: either strict First-In-First-Out (`FIFO`) ordering is maintained, or no ordering is guaranteed (`NONE`), as illustrated in Figure 3.3.

Figure 3.3: Lattice of ordering guarantees.

This relationship forms a simple two-element total order. A strict FIFO guarantee restricts the set of admissible system executions to only those that perfectly preserve sequence. A system expecting no particular order (NONE) will still function correctly even if messages arrive in strict sequence. Therefore, FIFO safely subsumes NONE. Formally, we define this as:

> Let $O = \{\text{NONE}, \text{FIFO}\}$, representing increasing levels of ordering guarantees. The total order is defined as:
>
> $$\text{NONE} \leq_O \text{FIFO}$$

Messaging Model

The messaging model defines the interaction pattern between producers and consumers. The two primary models supported are Point-to-Point (P2P), where each message is consumed by a single isolated consumer, and Publish/Subscribe (PUBSUB), where messages are broadcast to multiple distinct subscribers.

Figure 3.4: Lattice of (incomparable) messaging model guarantees.

Unlike delivery and ordering semantics, the messaging model does not form a total order; the models are fundamentally incomparable due to differing architectural semantics, as shown in Figure 3.4. A Pub/Sub system cannot strictly subsume a P2P system (nor vice versa) without application-level interventions. While an engineer might emulate a P2P queue by configuring a Pub/Sub topic with only a single subscriber, this requires manual architectural workarounds and does not reflect inherent system compatibility. Therefore, a mismatch in this dimension cannot be automatically resolved by the compatibility engine and will be flagged. Formally:

> Let $M = \{\text{P2P}, \text{PUBSUB}\}$, representing point-to-point and publish-subscribe messaging models. Because these are considered incomparable, we define:
>
> $$\text{P2P} \not\leq \text{PUBSUB} \land \text{PUBSUB} \not\leq \text{P2P}$$

Queue Guarantee Space

To properly evaluate a message queue, we aggregate its individual semantic dimensions into a unified mathematical space. A specific queue system is formally modelled as a tuple combining its delivery semantics, ordering guarantees, and messaging model:

$$q = (d, o, m) \in D \times O \times M$$

To determine whether a target queue (q_2) can safely replace a source queue (q_1), the compatibility engine evaluates the partial order component-wise. The formal definition is as follows:

$$q_1 \leq q_2 \iff d_1 \leq_D d_2 \land o_1 \leq_O o_2 \land m_1 = m_2$$

Therefore, a queue system is considered stronger if it provides stronger delivery and ordering guarantees while maintaining the same messaging model.

3.3.2. Key-Value Store Guarantees

Key-value stores are formally characterised across four critical dimensions: consistency, conditional writes, idempotence, and replication. By modelling these operational capabilities as partially ordered sets, we can systematically evaluate the safety of migrating an application from one storage backend to another.

Consistency

The consistency model dictates how and when updates to the data store become visible to subsequent read operations across the distributed system. Within this framework, consistency forms a simple two-element lattice, seen in Figure 3.5.

Figure 3.5: Lattice of consistency guarantees.

At the lower end of this lattice is eventual consistency. Systems employing eventual consistency prioritise high availability and partition tolerance. In this case, they guarantee that, in the absence of new updates, all replicas will eventually converge to the same state. However, they permit stale reads in the interim. At the upper end is strong consistency, which ensures that once a write operation is acknowledged, any subsequent read operation will return that updated value, regardless of which replica is queried.

This relationship forms a strict hierarchy. An application designed to tolerate the stale reads inherent to eventual consistency will continue to function perfectly well on a strongly consistent target system. Conversely, migrating an application that relies on strong consistency to an eventually consistent target introduces severe visibility gaps that can lead to critical synchronisation errors. Therefore, strong consistency safely dominates eventual consistency. Formally:

Let $C = \text{EVENTUAL}, \text{STRONG}$, with:

$$\text{EVENTUAL} \leq_C \text{STRONG}$$

Conditional Writes, Idempotence and Replication

The remaining three semantic dimensions represent specific operational capabilities. Because they denote the presence or absence of a feature, they are modelled as distinct Boolean lattices where the possession of the capability (TRUE) strictly dominates its absence (FALSE), as seen in Figure 3.6.

Figure 3.6: Boolean guarantee lattice.

The dimensions are as follows:

- **Conditional Writes**: This dimension captures whether the system supports atomic, predicate-based updates. While real-world systems implement this with varying levels of expressiveness, ranging from basic SETNX (Set if Not eXists) operations in Redis to complex attribute evaluation in DynamoDB and lightweight transactions (LWT) in Cassandra. This framework abstracts

these specific mechanics into a single boolean dimension representing the baseline presence or absence of atomic conditional capabilities.

- **Idempotence**: This property guarantees that repeatedly applying the exact same write operation will safely yield the same system state without producing unintended, cumulative side effects.
- **Replication**: This indicates the presence of built-in mechanisms for data redundancy, which are essential for fault tolerance and high availability.

In the context of system compatibility, an application built on a source system that lacks one of these capabilities (`FALSE`) makes no assumptions about its availability. If that application is migrated to a target system that provides the feature (`TRUE`), the migration is safe, as the excess capability is simply unused by the application logic. However, if an application depends on a capability like conditional writes for concurrency control, migrating to a target that lacks this feature (`FALSE`) will inevitably result in lost updates or data corruption. Thus, for all three Boolean properties, we define the partial orders for each dimension such that possessing a capability (`TRUE`) strictly dominates lacking it (`FALSE`). Formally:

Let W, I, and R be the Boolean domains representing the support for conditional writes, idempotence, and replication, respectively, where each domain is defined as $\{\text{FALSE}, \text{TRUE}\}$. We define the orders as:

$$\text{FALSE} \leq_W \text{TRUE}, \quad \text{FALSE} \leq_I \text{TRUE}, \quad \text{FALSE} \leq_R \text{TRUE}$$

Key-Value Guarantee Space

To evaluate the complete semantic profile of a key-value store, we aggregate these independent dimensions. A complete key-value store system is modelled mathematically as a tuple combining its specific guarantees:

$$k = (c, w, i, r) \in C \times W \times I \times R$$

To determine whether a target key-value store (k_2) is compatible with a source key-value store (k_1), the partial order is evaluated component-wise. A target system is considered stronger or equivalent only if it satisfies or exceeds the source system's constraints in every dimension. Formally:

$$k_1 \leq k_2 \iff c_1 \leq_C c_2 \land w_1 \leq_W w_2 \land i_1 \leq_I i_2 \land r_1 \leq_R r_2$$

3.3.3. System Guarantees and Compatibility Definition

To reason about the end-to-end compatibility of such distributed architectures, a complete system is defined as a pair uniting both queue and key-value store guarantees:

$$S = (q, k)$$

where q represents the queue guarantee tuple and k represents the key-value store guarantee tuple. To establish a safe migration path for the entire architecture, we define the partial order over these composed systems component-wise:

$$S_1 \leq S_2 \iff q_1 \leq q_2 \land k_1 \leq k_2$$

This construction forms a partially ordered set (poset) in which each dimension contributes independently to the overall ordering on the Cartesian product. Figure 3.7 visually maps this Cartesian product, illustrating how the independent partial orders of delivery, ordering, consistency, and boolean properties interact to form a unified guarantee space.

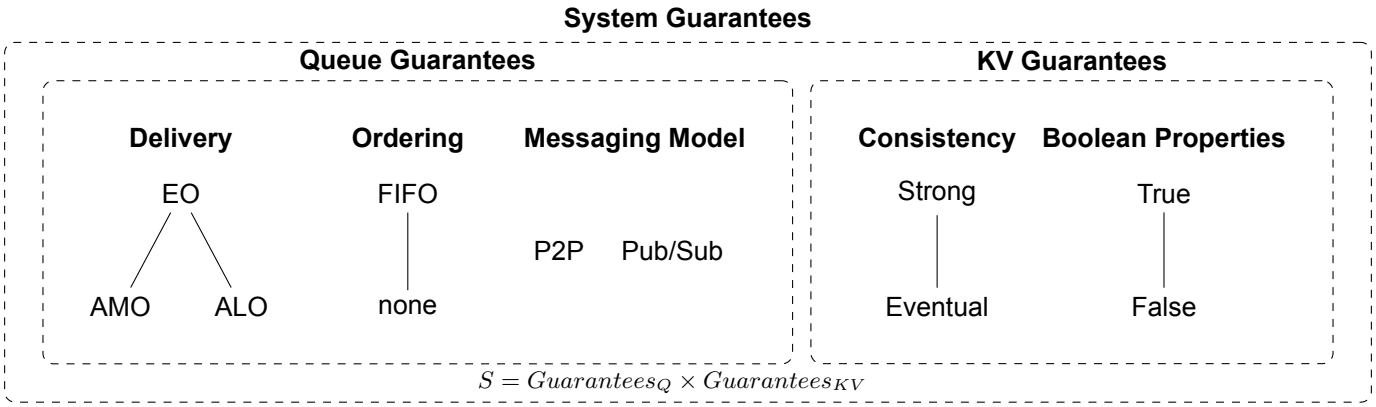


Figure 3.7: Overview of guarantee posets across all system dimensions. Each dimension forms a partially ordered set; system guarantees are defined as their Cartesian product.

By treating the composed system as a product poset, we ensure that the entire architecture adheres to the fundamental mathematical properties of a partial order:

- **Reflexivity**: $x \leq x$
- **Antisymmetry**: $x \leq y \land y \leq x \Rightarrow x = y$
- **Transitivity**: $x \leq y \land y \leq z \Rightarrow x \leq z$

Additionally, totally ordered dimensions (such as ordering and consistency) and subsets that form sub-lattices (such as delivery) admit well-defined join (least upper bound) and meet (greatest lower bound) operations, enabling reasoning about least common guarantees and maximal shared capabilities. This way, instead of relying on ad-hoc benchmarking or empirical testing, architectural substitutability is reduced to a machine-checkable comparison within a structured mathematical space.

Compatibility

To reason about compatibility between distributed systems, we model system guarantees as elements of a structured partially ordered space. This allows us to formally define when one system can safely substitute another based on its guarantees.

Using this formalisation, we define compatibility as follows:

> A target system is compatible with a source system if and only if it provides guarantees that are at least as strong in all dimensions.

Formally, given systems S_1 and S_2:

$$S_1 \leq S_2 \iff \forall d \in \text{dimensions} : S_1(d) \leq S_2(d)$$

This definition enables systematic reasoning about substitutability across heterogeneous systems.

This formalisation allows us to reason about substitutability in a principled manner. Instead of relying on system-specific behaviour, compatibility is reduced to a comparison in a structured guarantee space. This abstraction is particularly useful for evaluating portability across heterogeneous cloud systems, where guarantees determine correctness.

4

Experiments

This chapter presents the experimental evaluation of the proposed system for modelling and analysing cloud service semantics. The goal of these experiments is to assess the approach's correctness, expressiveness, and practical applicability across multiple dimensions.

The evaluation is structured around three main aspects. First, we investigate whether the framework can correctly determine compatibility between systems based on formally defined guarantees. Second, we evaluate the fidelity of the generated formal models by comparing their behaviour against a concrete system implementation using trace-based validation. Third, we explore the role of formal specifications in guiding LLM-based synthesis of distributed systems, focusing on whether invariant-based feedback improves correctness.

To support this evaluation, a series of experiments is conducted using automatically generated formal specifications, synthesised invariants, and execution traces derived from the Quint models. The experiments combine formal verification techniques with practical system execution, enabling a comprehensive assessment of both theoretical correctness and real-world behaviour.

The remainder of this chapter is organised as follows. Section 4.1 (Experimental Setup) describes the general experimental setup, including the tools, implementation details, and evaluation environment. Section 4.2 (System Compatibility Experiment) shows an experiment to determine compatibility between the chosen systems. Section 4.3 (Trace-Based Evaluation via ITF Parsing and Execution) presents the trace-based evaluation approach, including trace parsing, execution, and invariant validation. Finally, Section 4.4 (Evaluating the Impact of Formal Specifications on LLM-Based Synthesis of Distributed Systems) describes the experimental design for evaluating LLM-based synthesis.

4.1. Experimental Setup

To ensure the reproducibility of the formal verification and trace-based validation results, all experiments were conducted in a standardised environment. The hardware specifications provided the necessary computational capacity for simulation-based analysis, while specific software versions were fixed to maintain consistency across the generation and execution pipelines.

Hardware and Software Environment

The primary execution environment used a **2.8 GHz Quad-Core Intel Core i7** processor with **16 GB of 2133 MHz LPDDR3 RAM**. These specifications ensure that the exploration of the state space remains computationally tractable within reasonable time frames. The software stack is defined as follows:

- **Quint (v0.32.0):** Used as the core specification language to model cloud service semantics.
- **Python (v3.14.2):** Serves as the implementation language for the generative pipeline, configuration parsing, and trace replay logic.
- **Redis Server (v8.6.2):** Utilised as the concrete distributed system for verifying model fidelity during trace-based evaluation.

- **macOS Sequoia (v15.7.4):** Used as operating system.

General Experiment Configuration
The following parameters were applied to the randomised trace generation and system modelling phases to ensure statistical significance and broad coverage of potential system states:

- **Number of Samples:** A randomised pipeline generated 10,000 distinct execution traces to provide a diverse set of interleavings and operation patterns for validation. This is the default value defined by Quint, which led to its selection for these experiments.
- **Trace Lengths:** For large-scale randomised testing, traces were capped at 1,000 steps to balance exploration depth with simulation performance. Originally, the default value was 20 steps, but recent speed-ups made longer traces possible in these experiments.
- **Number of Replicas:** The distributed key-value store models were configured with three replicas to effectively evaluate synchronisation semantics and invariant preservation across divergent states. This value was deemed large enough to avoid overcomplication while still allowing proper testing of the replica feature. For systems that do not support replicas (like Memcached), this parameter was not taken into account.

Any deviations from these base parameters or experiment-specific configurations, such as the long-running 1,000-step trace evaluation, are detailed in their respective subsections.

4.2. System Compatibility Experiment

To systematically evaluate the friction points inherent in cross-cloud data migrations, we designed a pairwise compatibility experiment. The objective was to formally verify whether the specific semantic guarantees expected by an application connected to a `source` system would be preserved if that application were migrated to a `target` system.

4.2.1. Experimental Setup

To establish the experimental parameters, we defined a base specification for each domain (BaseQueue and BaseKV). These base specifications outlined the fundamental operations of the system types (e.g., enqueue and dequeue for queues, put and get for KV stores) without enforcing any strict semantic guarantees beyond basic data consistency.

We then evaluated the specific concrete systems and their representative operational configurations as previously defined in Chapter 2 (Background Information). This includes:

- **Message Queues**: Kafka, Pulsar, RabbitMQ, ActiveMQ, SQS, SQS FIFO, RedisQueue, and RocketMQ (configurations detailed in Table 2.1).
- **Key-Value Stores**: DynamoDB, Cassandra, MongoDB, CosmosDB, Redis, and Memcached (configurations detailed in Table 2.2).

4.2.2. Methodology

The experiment utilised an $N \times (N-1)$ pairwise evaluation approach for each domain, deliberately excluding same-to-same comparisons. For every possible combination of Source (S) and Target (T), the pipeline executed the following isolated procedure:

- **Configuration Generation**: An automated configuration was generated defining a composed system (`KVQueue`) where S acted as the source data store and T acted as the target.
- **Specification Synthesis**: The pipeline invoked a custom Python generator to synthesise a complete .qnt file. Crucially, the generator parsed the formal definition of S to extract its specific invariants (e.g., exactly_once, fifo_ordering). These source-specific invariants were appended to the standard base invariants required for system correctness.
- **Formal Verification**: The Quint CLI was invoked in an isolated environment to execute a randomised test run against the synthesised specification. The verification engine was instructed to explicitly check the combined list of standard and source-specific invariants.

- **Result Parsing**: If the Quint execution returned a zero exit code, the migration path from S to T was marked as ”compatible” (Pass). If the execution returned a non-zero exit code indicating a violation, the standard output was parsed to extract the exact name of the failed invariant.

The experiment yielded two JSON matrices mapping each source system to each target system, categorised as either a successful migration or a specific, localised formal verification failure.

4.3. Trace-Based Evaluation via ITF Parsing and Execution

To evaluate whether formally specified system behaviours correspond to real-world implementations, this work introduces a trace-based evaluation method. This method leverages execution traces generated by the Quint model checker and replays them on a concrete system (in this case, a Redis-based key-value store), validating invariants at each step. The pipeline is shown in Figure 4.1.

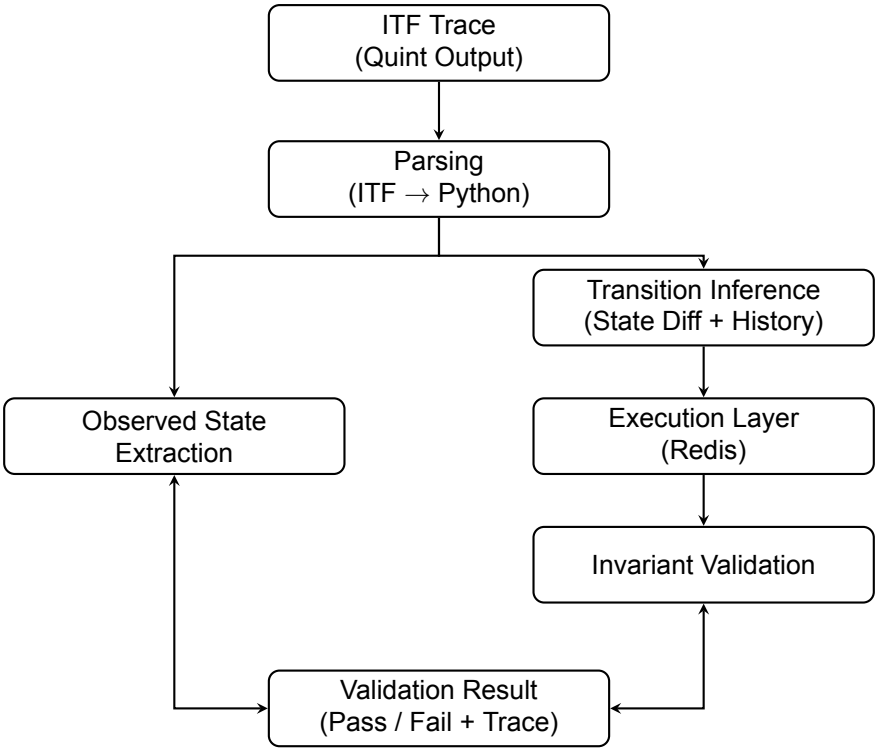


Figure 4.1: Trace-based evaluation pipeline.

4.3.1. Trace Representation

Traces are exported in the Informal Trace Format (ITF), defined in Apalache's ADR-015 [50]. ITF is a JSON-based format that encodes execution paths not as a sequence of explicit actions, but as a chronological sequence of full system states. Each trace contains a metadata block (`#meta`), a list of declared variables (`vars`), and the sequence of state snapshots (`states`).

To execute these abstract traces against a concrete system, the raw JSON states must be parsed into native Python data structures. ITF utilises special key prefixes to encode mathematical data types. Because these encodings are often deeply nested, for example, a `history` variable containing a set of structured message objects, the parsing process must be fully recursive to resolve the entire state tree.

Table 4.1 details how specific ITF encodings are translated into native Python types during this recursive parsing phase. Sets are parsed as a Python `list` rather than a `set` to accommodate non-hashable nested elements, such as dictionaries representing structured message records.

ITF Encoding	Formal Representation	Parsed Python Type
`#map`	Mapping of key-value pairs	`dict`
`#set`	Collection of elements	`list`
`#bigint`	Arbitrarily large integer	`int`

Table 4.1: ITF mathematical encodings and their corresponding parsed native Python types.

4.3.2. Inferring System Transitions

Since ITF traces do not explicitly encode actions, system transitions must be reconstructed by comparing consecutive states. This work introduces a hybrid approach that leverages both state differences in `kv` and semantic information in `history`.

The `history` variable plays a central role. It represents the set of active write operations (messages) in the system. By comparing the `history` sets between two consecutive states, it is possible to infer high-level operations such as writes, reads, syncs and deletions.

This approach provides a more accurate reconstruction of system behaviour than relying solely on differences in the key-value store, which can be ambiguous (e.g., distinguishing between writes and synchronisation events).

4.3.3. Execution Against Redis

Once transitions are inferred, they are executed against a real Redis instance. Each replica in the model is mapped to a Redis hash, where keys and values are stored as hash entries. The execution layer supports the following operations:

- **Write**: inserting a key-value pair into a replica
- **Delete**: removing a key from a replica
- **Sync**: propagating key-value pairs between replicas
- **Read**: modelled as a no-op, as it does not modify state

After each step, the state of the Redis instance is extracted and compared against the expected state from the trace. This ensures that the real system faithfully reproduces the behaviour specified by the model.

4.3.4. Invariant Validation

At each step of the replay, a set of invariants is evaluated on the observed system state. These invariants are derived from the Quint specification and capture essential correctness properties, such as:

- Absence of conflicting values across replicas
- Consistency between stored values and the history of writes
- Preservation of logical constraints under synchronisation and deletion

By validating invariants incrementally, the approach enables early detection of deviations between the model and the implementation.

The proposed method transforms abstract execution traces into concrete system executions, enabling systematic validation of distributed system behaviour. By combining structured trace parsing, operation inference, and real-system replay, it provides a practical mechanism for evaluating the correctness of cloud system implementations against their formal specifications.

4.4. Evaluating the Impact of Formal Specifications on LLM-Based Synthesis of Distributed Systems

This experiment investigates whether providing formal specifications improves an LLM's ability to synthesise a correct distributed key-value store. The main hypothesis is that formal specifications, ex-

pressed as invariants and system guarantees, act as a strong guidance signal during program synthesis. In particular, they may reduce the search space of possible implementations and accelerate convergence toward a correct solution.

To evaluate this hypothesis, we design an automated synthesis-and-verification loop in which an LLM iteratively generates and refines a key-value store implementation. The correctness of each generated implementation is evaluated using the same trace-based validation framework that was introduced in Section 4.3 (Trace-Based Evaluation via ITF Parsing and Execution). This framework serves as a correctness guide, providing structured feedback in the form of invariant violations.

4.4.1. Experimental Setup

The synthesis task is defined as implementing a distributed key-value store that supports multiple replicas. The system must correctly handle write, delete, and synchronisation operations while preserving a set of correctness properties derived from the formal specification.

To isolate the synthesis problem, the execution environment is fixed. Instead of interacting with an external system such as Redis, the LLM-generated implementation replaces the trace replay framework's execution backend. Specifically, the generated `kv_store.py` file implements the following interface:

```
class KVStore:
    def __init__(self, replicas): ...
    def write(self, replica, key, value, ts): ...
    def delete(self, replica, key): ...
    def get_state(self): ...
    def sync(self, replica): ...
```

The trace parser executes model-generated traces against this implementation and evaluates invariants after each step. If any invariant is violated, the system returns structured feedback describing the violation.

The synthesis process follows an iterative loop:

1. The LLM generates an initial implementation in response to a prompt.
2. The implementation is evaluated using trace replay and invariant checking.
3. If all invariants hold, the process terminates successfully.
4. Otherwise, feedback is generated and provided to the LLM.
5. The LLM produces a revised implementation based on the feedback.

All generative tasks within this synthesis loop are powered by Google's `gemini-3-flash-preview` model, accessed programmatically via the Gemini CLI. This specific model was selected for its optimised balance between rapid inference speed and advanced reasoning capabilities. This loop continues until a correct implementation is found or a maximum number of iterations is reached. To assess the impact of formal specifications, we define three experimental conditions:

1. **Full Specification**: The LLM is provided with a detailed description of system properties, including invariants such as absence of conflicts, consistency with write history, and correct synchronisation semantics.
2. **Natural Specification**: The LLM is given only a high-level description of the system and its API, in natural language, without explicit invariants or guarantees.
3. **No Specification (Baseline)**: The LLM is instructed to implement a generic distributed key-value store without additional guidance.

All other aspects of the setup, including prompts, iteration limits, and evaluation procedures, are kept constant across conditions to ensure comparability. The exact prompts used can be found in Appendix C (LLM-Based Synthesis Experiment Prompts).

To establish a rigorous evaluation, the maximum number of iterations should be carefully calibrated to find an experimental "sweet spot." If the iteration limit is set too low, the LLM is not afforded enough conversational turns to effectively process and leverage the invariant-based feedback, potentially resulting in zero convergence across all test groups. Conversely, if the limit is set too high, the model

might eventually stumble upon a correct implementation through sheer brute-force trial and error. A high limit obscures the measurable difference in guidance quality between the ”Full Specification” and ”No Specification” conditions. For this experiment, the iteration limit is capped at 100. This threshold is expected to be sufficient for an efficiently guided LLM to converge, while remaining restrictive enough to penalise unguided, random guessing.

4.4.2. Evaluation Metrics

The effectiveness of each condition is evaluated using the following metrics:

- **Iterations to Convergence**: The number of iterations required to produce a correct implementation that satisfies all invariants.
- **Code Churn**: The number of modifications to the code during synthesis.
- **Time to Convergence**: The total time required to reach a correct implementation.
- **Tokens per Full Run**: The total API output token volume consumed across the entire synthesis loop, serving as a proxy for computational load and financial cost.

These metrics capture both the efficiency and reliability of the synthesis process under different levels of specification.

4.4.3. Synthesis and Feedback Loop

The synthesis process operates as a black-box refinement loop. The LLM is not trained or fine-tuned during the experiment. Instead, it relies solely on the provided prompts and feedback to iteratively improve its output.

Feedback is generated by the trace validation framework in a structured format, indicating which invariants were violated and at which step. This feedback is then incorporated into the next prompt, allowing the LLM to correct specific errors. For example, a violation of the `lww_conflict_resolution` invariant may indicate that replicas diverge, prompting the model to adjust its synchronisation logic.

This approach ensures that improvements in the generated implementation are driven by the formal specification rather than implicit heuristics.

5 Results

This chapter presents the findings from the experiments outlined in Chapter 4 (Experiments), evaluating the efficacy and practical applicability of the proposed formal framework for cloud message queue and key-value store semantics.

The chapter is organised as follows. Section 5.1 details the outcomes of the pairwise system compatibility experiments, providing a comprehensive mapping of semantic friction points across both message queues and key-value stores. Section 5.2 presents the findings from the trace-based validation, assessing whether the abstract behaviours defined in the Quint models accurately align with the concrete execution of a real Redis instance. Finally, Section 5.3 reports on the usability of these formal specifications within an LLM-based system generation loop, evaluating the extent to which invariant-based feedback improves the efficiency of synthesising correct LLM-synthesised distributed systems. Together, these results provide a quantitative and qualitative foundation for the broader discussion in the subsequent chapter.

5.1. System Compatibility

To evaluate the cross-system compatibility of the modelled architectures, we executed a pairwise invariant verification experiment. Each system was sequentially assigned as a source and mapped to every other system acting as a target. The formal verification engine assessed whether the target system could uphold all data invariants expected by the source system. The results are divided into message queues and KV stores.

5.1.1. Message Queue Compatibility

Message queue systems exhibited a highly fragmented and restrictive compatibility landscape. A precise system-by-system breakdown of these barriers is provided in Table 5.1.

Source ↓ / Target →	ActiveMQ	Kafka	Pulsar	RabbitMQ	RedisQueue	RocketMQ	SQS	SQS_FIFO
ActiveMQ	-	✓	✓	✓	✓	✓	✓	✓
Kafka	*no_processing_after_ack*	-	✓	✓	*no_processing_after_ack*	*no_processing_after_ack*	*no_processing_after_ack*	✓
Pulsar	*no_processing_after_ack*	✓	-	*fifo_ordering*	*no_processing_after_ack*	*no_processing_after_ack*	*no_processing_after_ack*	✓
RabbitMQ	*at_most_once*	✓	✓	-	*at_most_once*	*at_most_once*	*at_most_once*	✓
RedisQueue	*fifo_ordering*	✓	✓	*fifo_ordering*	-	*fifo_ordering*	*fifo_ordering*	✓
RocketMQ	✓	✓	✓	✓	✓	-	✓	✓
SQS	✓	✓	✓	✓	✓	✓	-	✓
SQS_FIFO	*no_processing_after_ack*	✓	✓	*fifo_ordering*	*no_processing_after_ack*	*no_processing_after_ack*	*no_processing_after_ack*	-

Table 5.1: Compatibility matrix for message queue systems showing invariant violations. ✓ denotes full compatibility.

The Queue Compatibility Heatmap in Figure 5.1 highlights a dense distribution of minor incompatibility with 1 invariant violation (yellow blocks) and fully compatible systems (green blocks) across the matrix, indicating that queues often only fail on a single semantic dimension.

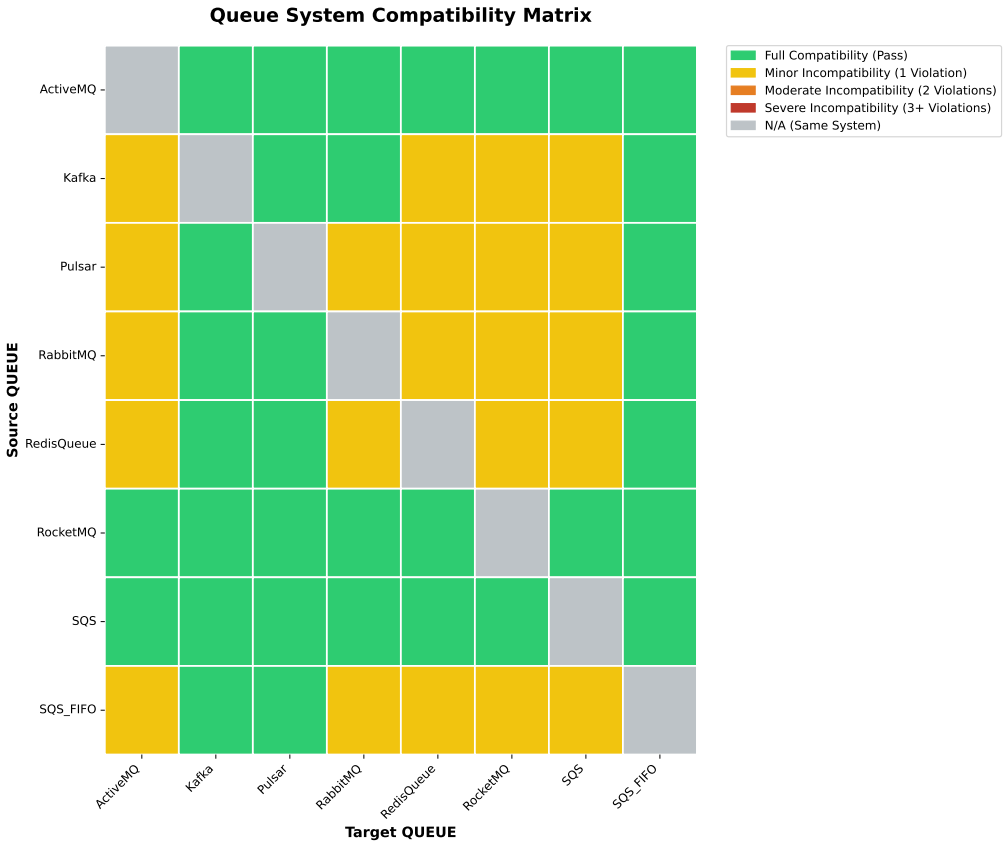


Figure 5.1: Heatmap of queue compatibility

To isolate these constraints, the compatibility network was decomposed into passing (Figure 5.3) and failing (Figure 5.2) pathways. Although we reason about the two figures separately, a combined graph of all pathways is provided in Appendix E (Full Queue Network Graph). The ”Fail-Only” network (Figure 5.2) visually shows the network of failures, and in combination with Table 5.1, clusters of specific invariant violations responsible for breaking queue compatibility can be seen:

- `no_processing_after_ack`: Strict exactly-once sources (like Kafka, Pulsar, and SQS_FIFO) failed to migrate safely to looser at-least-once targets (such as ActiveMQ, RedisQueue, RocketMQ, and standard SQS) because the targets could not mathematically guarantee that a message would not be redelivered or re-processed after an initial acknowledgment.
- `fifo_ordering`: Sequence-dependent sources (such as RedisQueue, Pulsar, and SQS_FIFO) triggered violations when mapped to unordered or queue-level ordered targets (like RabbitMQ, ActiveMQ, RocketMQ, and standard SQS), as the targets could not uphold strict FIFO delivery rules.
- `at_most_once`: Migrations originating from systems configured with strict at-most-once delivery boundaries (ActiveMQ, RabbitMQ, RocketMQ, and standard SQS) generated violations when mapped to at-least-once targets. In practical terms, this means the source application was likely not designed to handle duplicate messages (e.g., it lacks idempotency). Thus, migrating to a target system that might deliver the same message multiple times fundamentally breaks the application’s correctness, leading to a verified incompatibility.

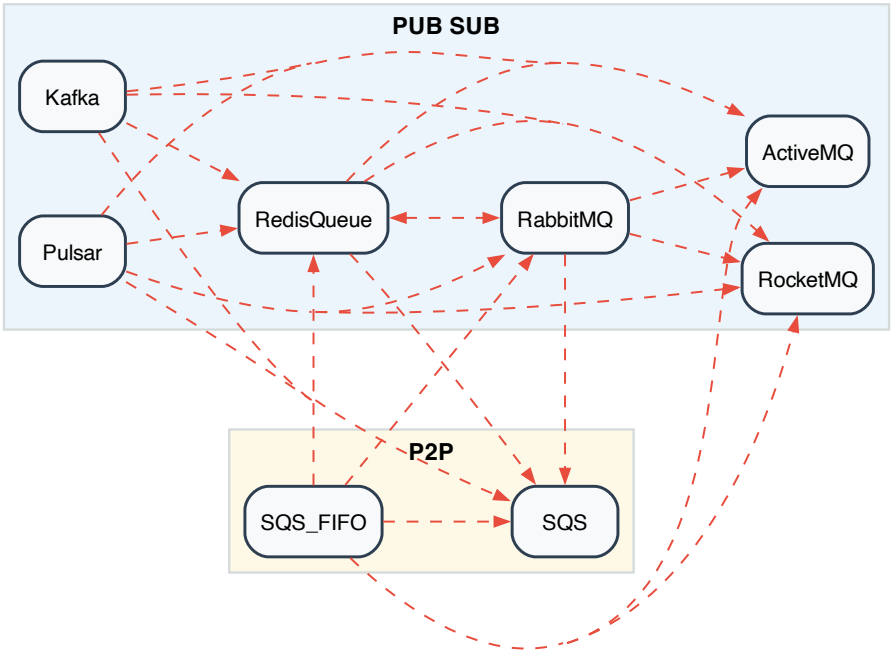


Figure 5.2: Network graph of queue compatibility failures

On the other hand, the ”Pass-Only” network (Figure 5.3) highlights the highly directional nature of queue compatibility. Permissive systems (e.g., ActiveMQ, SQS, RocketMQ) serve as universal sources, providing numerous safe outbound migration paths. In contrast, strict systems (e.g., Kafka, SQS_FIFO) function as sink nodes; they readily accept incoming migrations but offer almost zero safe outward paths without breaking invariants. While the network reveals multiple safe migration pathways, the presence of failure edges highlights that out-of-the-box substitutability in this ecosystem should be navigated with caution.

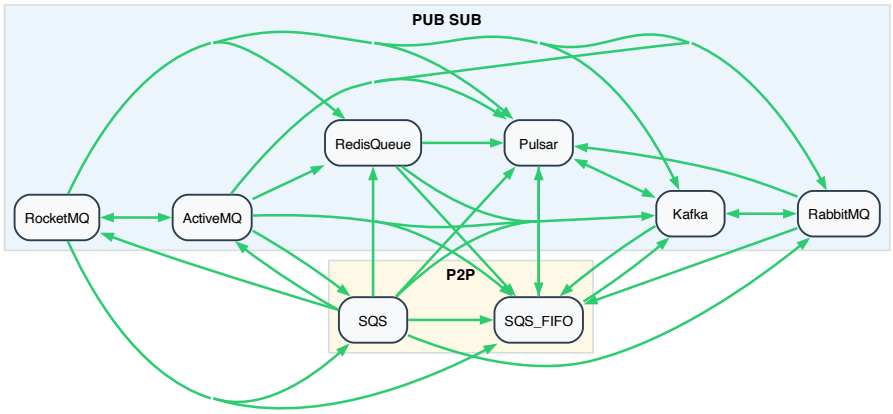


Figure 5.3: Network graph of queue compatibility successes

Ultimately, the message queue compatibility analysis demonstrates that queues are highly fragmented and restrictive compared to key-value stores. The presence of strict operational constraints, specifically regarding delivery boundaries and sequence dependencies, drastically limits seamless interoperability. Systems that enforce robust guarantees essentially act as migration sink nodes, meaning that over-provisioning these semantics directly increases the risk of vendor lock-in. Consequently, when organisations attempt to migrate away from such strict systems, they are often met with hard structural lock-in, forced to either accept significant semantic downgrades or undertake application-level rewrites to compensate.

5.1.2. Key-Value Store Compatibility

In contrast to queues, the compatibility landscape for KV stores is predominantly permissive, characterised by a high degree of interoperability with isolated, specific semantic failures. As illustrated visually in the KV Compatibility Heatmap (Figure 5.4) and detailed precisely in Table 5.2, a majority of the source-to-target migrations successfully preserved system invariants.

Source ↓ / Target →	Cassandra	CosmosDB	DynamoDB	Memcached	MongoDB	Redis
Cassandra	-	✓	*causality*	✓	*causality*	*causality*
CosmosDB	✓	-	*causality*	✓	*causality*	*causality*
DynamoDB	✓	✓	-	✓	✓	✓
Memcached	✓	✓	✓	-	✓	✓
MongoDB	✓	✓	✓	✓	-	✓
Redis	✓	✓	✓	✓	✓	-

Table 5.2: Compatibility matrix for key-value stores showing invariant violations.

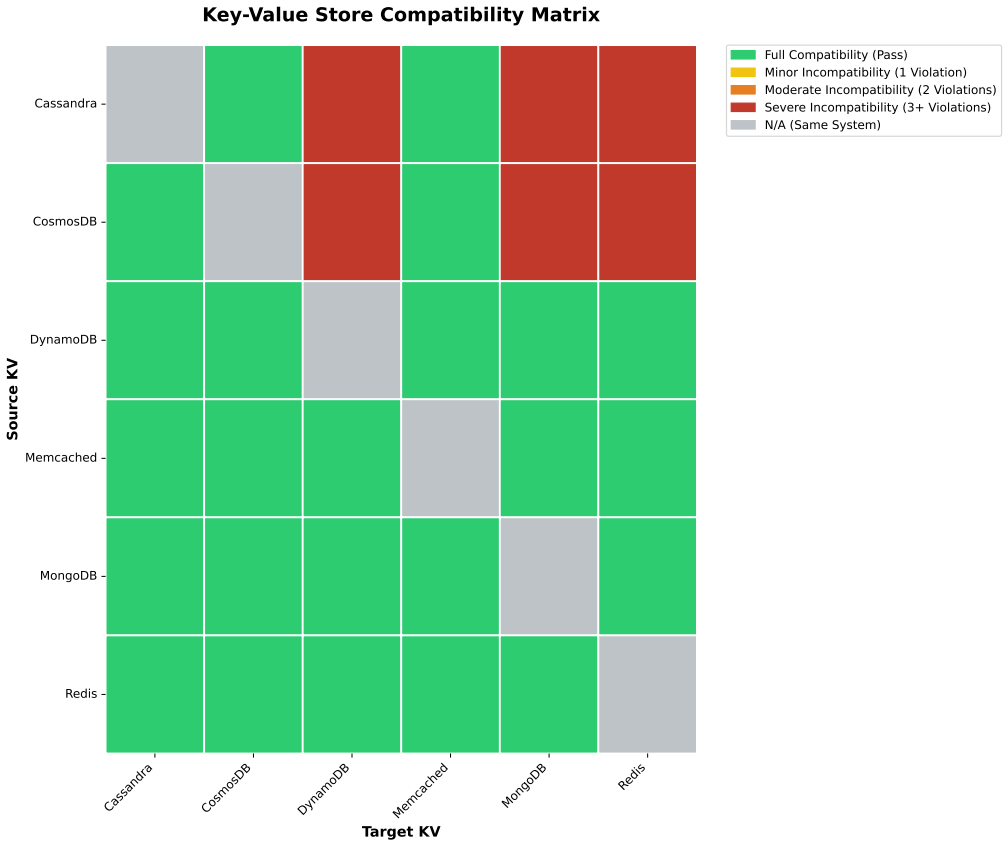


Figure 5.4: Heatmap of KV compatibility

However, the heatmap also reveals a highly specific, clustered bottleneck: the red squares in the matrix indicate severe incompatibilities (3+ violations). When source systems configured with strong-consistency requirements (Cassandra, CosmosDB) or immediate-visibility assumptions (Memcached) are migrated to target systems that utilise eventual consistency or asynchronous replication (DynamoDB, MongoDB, Redis), the verification engine detects a massive cascade of invariant failures. Because eventual consistency allows ”windows of vulnerability” in which replicas temporarily diverge, these targets cannot mathematically guarantee strict causal ordering or immediate state synchronisation, resulting in multiple violations rather than minor semantic friction.

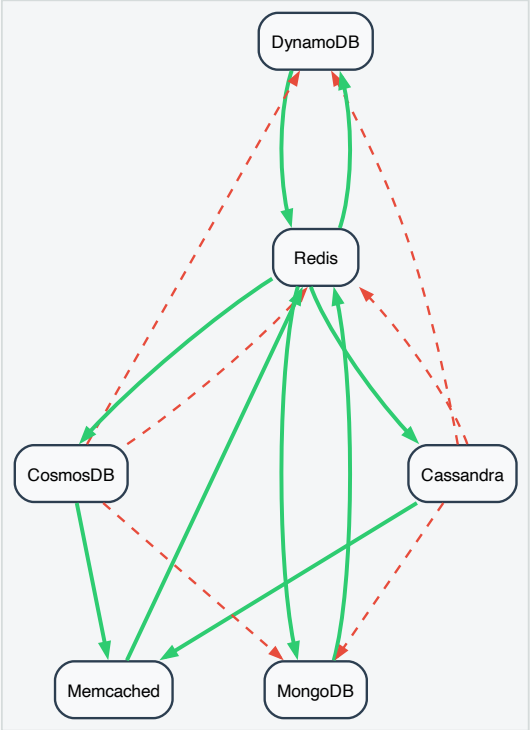


Figure 5.5: Network graph of KV compatibility

This permissive but directional nature is further illustrated in the KV Compatibility Network Graph (Figure 5.5). The directed edges map the flow of safe migrations. Notably, systems like DynamoDB and MongoDB exhibit excellent mobility as sources (producing green outward edges) because their loose eventual-consistency guarantees are easily accommodated by stricter targets. Conversely, when attempting to use these eventually consistent systems as targets for strict sources, the migrations fail (represented by the red edges pointing to DynamoDB, MongoDB, and Redis), highlighting a clear semantic boundary in data store migration.

5.1.3. Composition Compatibility

To evaluate the interaction of end-to-end architectures, we extended the formal verification to the composed system models ($S = Guarantees_Q \times Guarantees_{KV}$ mentioned in Section 3.3 (Formalization of System Guarantees)). This experiment assesses whether an entire application that depends on both a specific message queue and a specific key-value store can be safely migrated to a new combined stack.

The Composition Compatibility Heatmap (Figure 5.6) visualises the Cartesian product of all evaluated queues and key-value stores. Compared with the individual system matrices, the composite heatmap shows a significant decrease in the number of fully compatible blocks (green blocks). Because the compatibility partial order is evaluated component-wise ($S_1 \leq S_2 \iff q_1 \leq q_2 \wedge k_1 \leq k_2$), a failure in either the queue layer or the KV layer immediately flags the entire migration as incompatible.

Consequently, the resulting matrix is populated with a spectrum of failures, ranging from minor (yellow) to severe (red). Because the evaluation is component-wise, the invariant violations are effectively additive. A minor incompatibility in the queue layer (1 violation) combined with a moderate incompatibility in the storage layer (2 violations) results in a red semantic gap. This highlights that safe, end-to-end cloud migrations are extremely restrictive; a downgrade in either the database or messaging layer will structurally break the application across multiple dimensions.

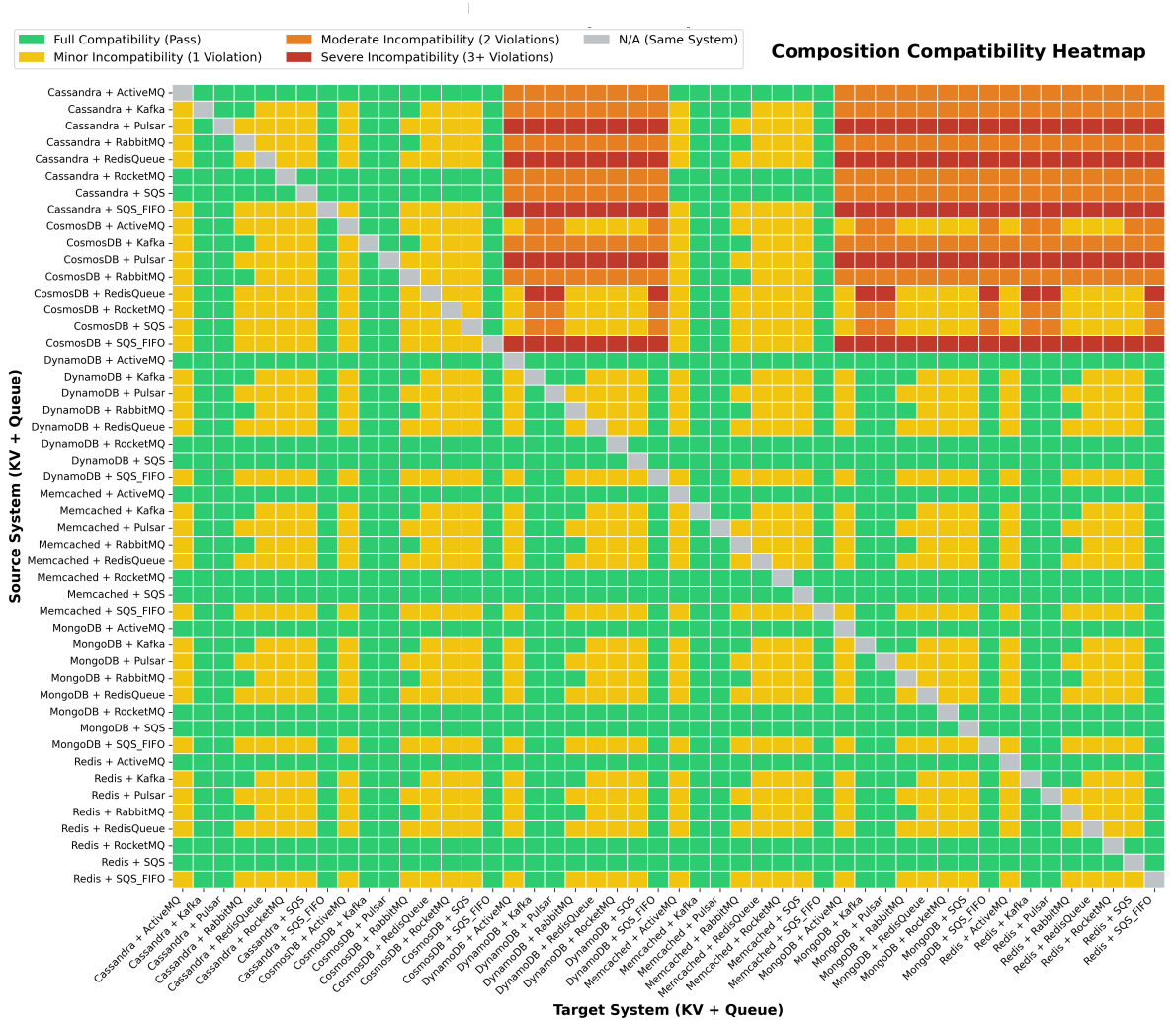


Figure 5.6: Heatmap of composition compatibility

Migration Mobility: Compatibility Score per Source

To quantify how easily a specific composed architecture can be migrated away from, we calculated a Compatibility Score per Source (Figure 5.7). This metric represents the percentage of available target combinations a given source can successfully migrate to.

The results clearly illustrate that ”weaker” composed systems possess the highest migration mobility. Highly permissive sources combining eventual consistency with best-effort or at-least-once messaging (e.g., MongoDB + ActiveMQ, MongoDB + RocketMQ) scored the highest, successfully mapping to 100% (1.0) of all potential targets. The next tier of moderately permissive systems successfully mapped to approximately 50% of targets. Because these architectures make very few strict assumptions about data visibility or message ordering, they can safely operate on a wide variety of target platforms. Conversely, strict sources (e.g., Cassandra + Pulsar, CosmosDB + Kafka) scored under 20%, confirming that applications built on highly rigid, exactly-once, and strongly consistent foundations suffer from strong semantic vendor lock-in.

Target Robustness: Capability to Accept Migrations

Conversely, we evaluated how effectively specific composed architectures can act as migration targets (Figure 5.8). Target Robustness is defined as the system’s ability to safely absorb incoming migrations by satisfying the required source invariants.

The data reveal an inverse relationship with source mobility, where the strictest systems yield the most

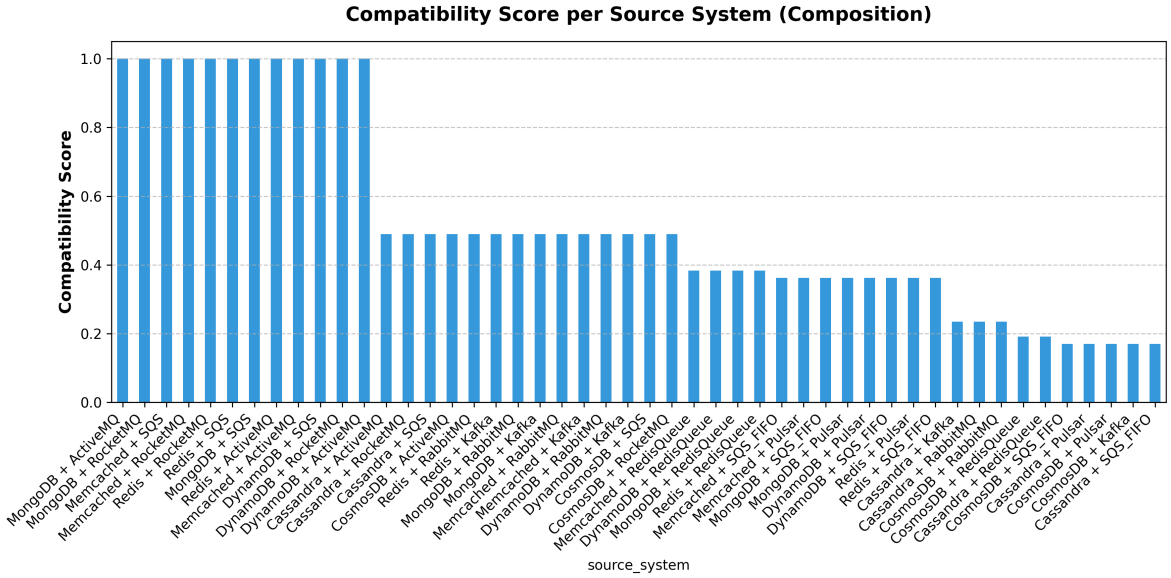


Figure 5.7: Composition scores per source

robust targets. Target architectures that pair exactly-once, FIFO-ordered queues with strongly consistent key-value stores (e.g., Cassandra + Kafka, CosmosDB + Pulsar) achieved perfect or near-perfect robustness scores of 1.0. Because these systems enforce the absolute maximum level of mathematical guarantees, they effortlessly subsume the weaker invariants of incoming applications. Meanwhile, highly permissive systems (like Redis + ActiveMQ or DynamoDB + SQS) scored poorly as targets, routinely rejecting incoming migrations because they could not uphold the stricter requirements expected by the source applications.

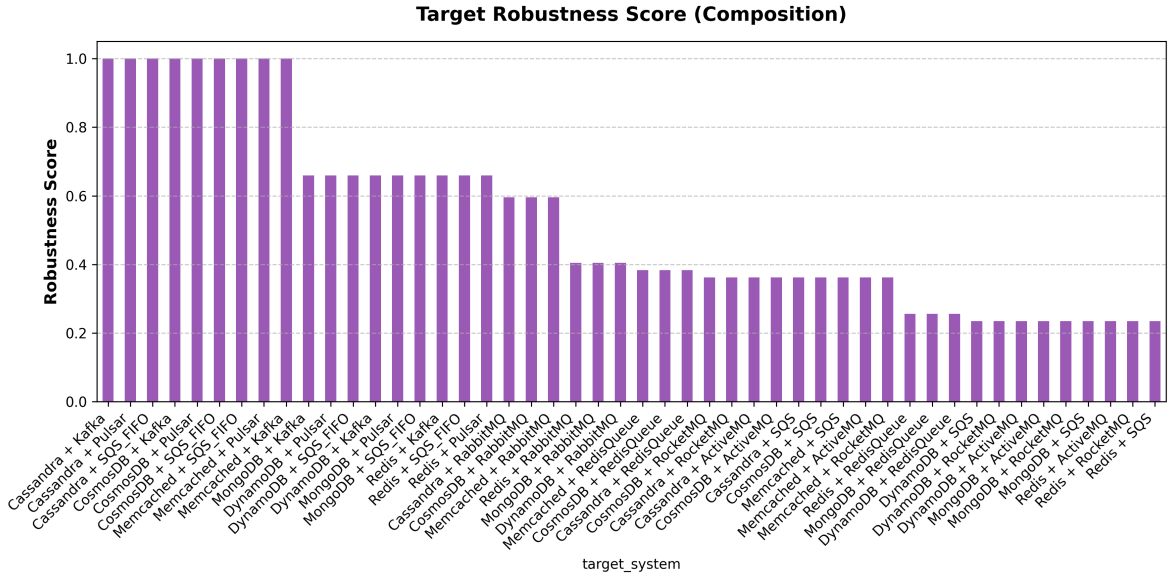


Figure 5.8: Composition scores per target

Failure Distribution in Composed Systems

Finally, an analysis of the root causes for composition failures (Figure 5.9) demonstrates that migration barriers are almost equally split between the database and messaging layers.

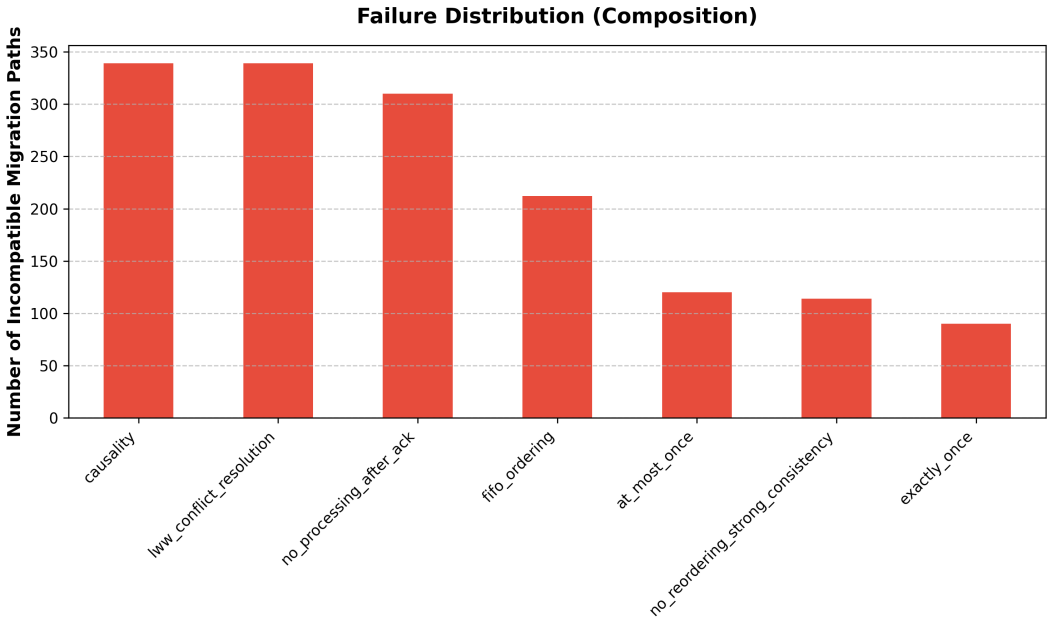


Figure 5.9: Failure distribution

The two most frequent invariant violations were KV-specific: causality and lww_conflict_resolution, both of which triggered over 300 migration failures. This indicates that mismatches in state synchronisation and replica convergence remain the most pervasive threats to end-to-end data integrity.

From a practical perspective, this distribution highlights exactly where architectural investments yield the highest return on mobility. For example, the overwhelming prevalence of causality violations suggests that if an engineering team retrofits their application logic to safely handle eventual consistency (effectively bypassing the strict causality requirement at the application layer), they immediately unlock compatibility with over 300 previously incompatible cloud service combinations. By systematically resolving these specific bottleneck violations, organisations can exponentially increase their migration mobility, transforming a highly locked-in architecture into a truly cloud-agnostic one.

However, queue-specific violations closely followed, with no_processing_after_ack triggering over 300 violations and fifo_ordering triggering over 200. This distribution shows that evaluating a cloud system's compatibility in isolation is insufficient; a holistic migration strategy must simultaneously resolve strictness mismatches in both data storage visibility and message delivery flow.

5.2. Trace-Based Validation of Key-Value Store Semantics

To evaluate the proposed trace-driven validation approach, we conducted a series of experiments in which execution traces generated by the Quint model were replayed against a real Redis instance. The goal of these experiments was to assess whether the behaviour of the concrete system conforms to the formally specified model and whether the defined invariants hold under realistic execution sequences.

Two complementary trace generation pipelines were used. First, a large-scale randomised pipeline generated 10,000 traces with a maximum length of 1000 steps. From this set, a representative trace was selected for replay and validation. This configuration ensures broad coverage of short execution patterns and diverse interleavings of operations. Second, a long-running pipeline generated 10 traces, each 100,000 steps long. This configuration enables the evaluation of system behaviour over extended execution sequences, capturing long-term effects such as repeated updates, deletions, and synchronisation patterns. After each step, the resulting Redis state is compared against the expected model state, and a set of invariants is evaluated.

5.2.1. Correctness of Trace Replay

Across both pipelines, all traces were successfully replayed without execution errors after resolving initial inconsistencies in state initialisation and transition inference. In particular, the explicit initialisation of replica states in Redis was required to align Redis's implicit representation of the empty state with the model's explicit state representation.

All evaluated invariants held across both short and long traces once the model semantics were aligned with the execution environment. In particular, invariants ensuring the absence of divergent values across replicas and the consistency between stored values and previously issued writes were satisfied throughout all executions, since the generated invariants corresponded with the eventual consistency of Redis.

Overall, the results demonstrate that the proposed trace-driven validation approach accurately reproduces model-generated execution traces on a real system and verifies their correctness using formally defined invariants. The approach is robust to different trace lengths and execution patterns and provides valuable insights into the alignment between abstract models and concrete system implementations.

Moreover, the experiments uncovered several subtle mismatches between the model and the system, including differences in state representation and operation semantics. Addressing these mismatches not only improved the validation framework's correctness but also strengthened the model's fidelity. These changes will be further discussed in Section 6 (Discussion).

5.3. Impact of Formal Feedback on System Synthesis

To establish statistical significance and account for the inherently probabilistic nature of Large Language Models, the automated synthesis loop was executed across multiple runs for each of the three experimental conditions. The results, visualised as box plots in Figure 5.10, reveal a nuanced landscape in which the level of specification profoundly affects not only the efficiency of the synthesis but also its predictability and computational cost.

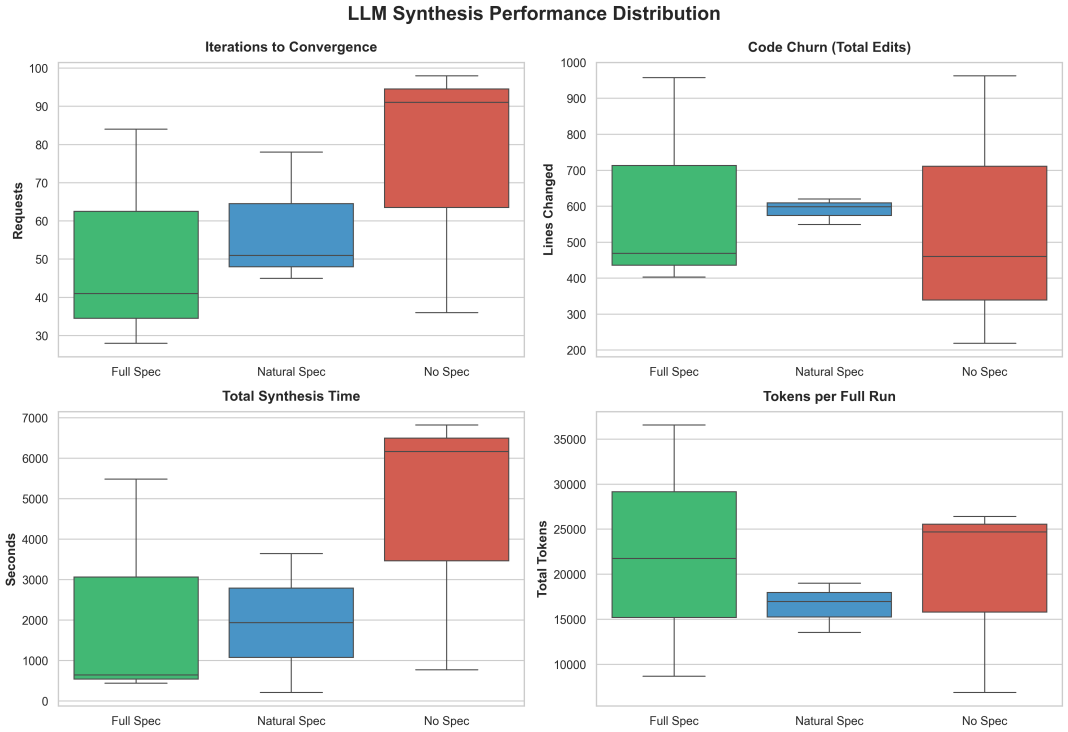


Figure 5.10: LLM Synthesis Results for Multiple Runs

Iterations and Code Churn

The distribution reveals that while formal guidance reduces the search space on average, it introduces significant volatility. The Full Specification condition achieved the lowest median number of iterations to convergence ($\sim$40), confirming that strict invariants generally help the LLM bypass incorrect logical pathways. However, its large variance (with upper whiskers extending beyond 80 iterations) indicates that when the LLM struggles to satisfy an invariant, it can easily become trapped in prolonged cycles of ineffective refinement.

Conversely, the Baseline (No Spec) condition proved to be highly erratic. While it had the highest median number of iterations ($\sim$95), its Code Churn distribution is deeply revealing. The immense spread of lines changed from a highly efficient $\sim$220 edits to a massive $\sim$960 edits, suggesting that without guidance, the LLM relies entirely on trial-and-error. In some runs, it achieves a ”lucky guess” early on; in others, it completely thrashes, rewriting the entire code multiple times without a clear target. Nestled between these extremes is the Natural Specification condition. While it requires slightly more iterations than the Full Spec (median $\sim$50), it is quite predictable, with the tightest variance across both iterations and code churn.

Time and Computational Cost

By evaluating both the Total Synthesis Time (in seconds) and Tokens per Full Run (Figure 5.10), we can assess the operational efficiency of each guidance method. The temporal data reveals a duality in the Full Specification condition. On average, it achieved the fastest median synthesis time (approximately 500 seconds). Because it requires the fewest iterations, it inherently reduces the number of network round trips to the LLM API. However, the extreme upper whiskers of the distribution, stretching to over 12,000 seconds, expose stark worst-case scenarios. When the LLM fails to quickly parse the formal invariants, the generation process stalls significantly. The Baseline (No Spec) was consistently the slowest (median $\sim$6,000 seconds), while the Natural Specification sat predictably in the middle (median $\sim$2,000 seconds).

However, wall-clock time is susceptible to network latency and API load. To accurately measure the financial and computational burden, we look to Tokens per Full Run. This deterministic metric clearly establishes the Natural Specification condition as the most efficient approach, maintaining a tightly bounded, low median of approximately 17,000 total tokens.

In contrast, the Full Specification proved computationally expensive and highly volatile (median $\sim$22,000, up to $\sim$36,000 tokens). This increase is likely driven by the fact that feeding dense logic and rigid invariant failures back into the LLM’s context window requires significantly more tokens per request. The Baseline (No Spec) condition, while having a smaller context, remained highly expensive (median $\sim$25,000 tokens) probably due to the sheer volume of trial-and-error iterations required. Ultimately, while formal specifications provide a direct path to correctness, lighter natural-language guidance currently offers the optimal balance: it avoids the time delays of formal math loops while delivering superior predictability and lower overall token consumption.

6

Discussion

This chapter discusses the results of the proposed experiments in relation to the research objectives (Section 1.2) and questions (Section 1.3), as well as the broader context of existing work on cloud interoperability and formal methods. The goal is to interpret the findings beyond their immediate experimental setting, identify trends and implications, and position this thesis's contributions within the current state of the art. First, the results of each experiment will be discussed separately, followed by general observations. Finally, the limitations of this thesis will be highlighted.

6.1. System Compatibility

The results of the pairwise compatibility experiment expose a fundamental reality of cross-cloud data migration: semantic compatibility is inherently asymmetrical. A successful migration is not determined by whether two systems are ”equivalent,” but rather by whether the target system's invariant guarantees act as a strict superset of the source system's requirements.

6.1.1. Semantic Lock-in

The directed nature of the compatibility graphs (Figures 5.3 and 5.5) visually confirms that safe migration pathways are rarely bidirectional. The flow of edges across these networks dictates that ”upgrading” the system during a migration is generally safe, whereas ”downgrading” the system introduces fatal structural flaws. Systems like Kafka and Pulsar act as strict origins; their internal invariants are so rigid that migrating away from them requires a target of equal or greater strictness, severely limiting architectural mobility.

These findings introduce a critical dimension to the concept of vendor lock-in. Historically, lock-in is discussed in terms of proprietary APIs, data formats, or egress costs. However, this experiment demonstrates *Semantic Lock-in*.

The compatibility heatmaps (Figures 5.1, 5.4 and 5.6) serve as maps of the depth of this lock-in. Instead of viewing lock-in as a simple binary pass/fail state, the results categorise the semantic gap by severity. Yellow and orange areas (minor to moderate incompatibilities) represent targeted semantic mismatches, in which an application might lose only a single guarantee, such as FIFO ordering. While these still mathematically violate correctness, they represent a smaller engineering hurdle to retrofitting. Conversely, the red areas denote severe, hard structural lock-in (3+ violations) where fundamental guarantees fail across multiple architectural dimensions simultaneously. In these instances, even if a developer successfully rewrites an application's API calls to shift from CosmosDB to DynamoDB, the system will likely fail unpredictably because the underlying formal guarantees are violated in multiple ways. Meanwhile, the green areas represent pathways of semantic freedom. However, as discussed in Section 6.5, even these safe pathways require careful validation of implicit application behaviours before executing a real-world migration.

6.1.2. Bottleneck Invariants

By analysing the specific failure distributions across the matrices, the formal verification system successfully isolated the bottleneck guarantees that drive Semantic Lock-in.

- **Key-Value Stores**: The `causality` invariant proved to be the ultimate barrier. Relying on strict state synchronisation limits a system's ability to migrate to highly available, eventually consistent data stores.
- **Message Queues**: The combination of `no_processing_after_ack`, `fifo_ordering`, and `at_most_once` invariants dictated queue mobility.

This suggests a vital principle for initial system design: engineers should avoid adopting systems with strict causality, FIFO ordering, or rigid acknowledgment semantics unless dictated by absolute business necessity. As the heatmaps visually confirm, over-provisioning semantic guarantees at the start of a project artificially narrows the pool of viable targets for future migrations, inadvertently and permanently anchoring the architecture to a specific cloud provider.

6.1.3. Composition Compatibility

The evaluation of composed architectures ($S = Guarantees_Q \times Guarantees_{KV}$) demonstrates that end-to-end migrations suffer from a multiplicative failure effect. Because compatibility is evaluated component-wise, a semantic mismatch in either the database or messaging layer invalidates the entire migration path. As seen in the Composition Compatibility Heatmap (Figure 5.6), full out-of-the-box compatibility becomes exceedingly rare in multi-component architectures.

The analysis reveals an inverse relationship between an architecture's migration mobility and its robustness as a target. "Weaker" composed systems, those pairing eventual consistency with at-least-once or best-effort delivery (e.g., MongoDB + ActiveMQ, Memcached + SQS), achieve the highest source compatibility scores. Because they make minimal strict assumptions about data visibility or sequencing, they can be safely mapped to a vast majority of cloud targets. Conversely, highly strict architectures (e.g., Cassandra + Kafka, CosmosDB + Pulsar) score poorly as sources but achieve perfect target robustness scores. They act as absolute migration sinks, readily absorbing the weaker invariants of incoming applications while permanently constraining the architecture from future outward migrations.

Finally, the failure distribution for composed systems (Figure 5.9) underscores that modern cloud lock-in cannot be analysed in isolated silos. Violations are driven almost equally by storage synchronisation limits (e.g., `causality`, `lww_conflict_resolution`) and messaging constraints (e.g., `no_processing_after_ack`, `fifo_ordering`). Therefore, escaping vendor lock-in requires a holistic, full-stack semantic analysis to ensure that the complex interplay between data storage visibility and message flow is fully verified prior to migration.

6.2. Trace-Based Validation and Model Fidelity

The trace-based evaluation results provide strong evidence for the fidelity of the formal models. The absence of invariant violations and the exact match between the expected and observed states indicate that the model captures the relevant aspects of the system's behaviour.

An important observation was made regarding the semantics of deletion. Initially, deletions were modelled as the removal of entries from the history. This led to invariant violations when data remained on other replicas due to delayed synchronisation. The issue was resolved by redefining history as an append-only structure, reflecting the fact that deletions do not erase past operations but instead represent new events. This change aligns the model more closely with real-world distributed systems, where deletions are typically implemented as tombstones.

Furthermore, the introduction of a history variable in the model enabled more accurate reconstruction of write and delete operations. However, it was observed that history alone is insufficient to infer all transitions, as operations such as synchronisation and deletion do not necessarily modify the history. This necessitated a combined approach that used both historical and state differences to correctly infer transitions.

In particular, delete operations were initially misclassified as synchronisation events, leading to incorrect

execution attempts. This issue was resolved by incorporating state-based differencing, which allows the system to distinguish among write, delete, and synchronisation operations based on changes in the key-value store.

This highlights that accurately reconstructing system behaviour from traces requires combining semantic information (history) with structural information (state differences). This finding is particularly relevant for systems where not all operations are explicitly logged or observable.

However, it is important to note that these results are limited to the evaluated scenarios. While both short and long traces were considered, the evaluation does not guarantee correctness under all possible conditions.

Future work could extend this evaluation by incorporating more diverse workloads, larger state spaces, and different system configurations. Additionally, applying the approach to multiple real-world systems would provide further evidence of its generality.

6.2.1. Consistency Analysis of Asynchronous Replication in Redis

Initial trace-based validations of the Redis model against concrete execution traces yielded unexpected invariant violations, specifically regarding strict causality and Last-Write-Wins (LWW) conflict resolution. However, a deeper analysis revealed that these failures were not due to an implementation flaw in the concrete Redis server or the Quint model, but rather inappropriate, over-constrained invariants applied to an asynchronously replicating system. For instance, the initial `causality` invariant demanded that a processed message be immediately visible across all replicas, which indeed should not hold with a system that is only eventually consistent. Once the invariants were corrected to properly reflect the capabilities of Redis, namely, as an eventually convergent key-value store, the traces executed successfully without any violations. This highlights a critical lesson in formal verification: generated invariants must precisely match the relaxed constraints of the target architecture to avoid false negatives during validation.

6.3. LLM-Based Synthesis and the Role of Formal Specifications

The preliminary design of the LLM-based synthesis experiment suggests that formal specifications can act as an effective guidance mechanism for program synthesis. By providing explicit correctness criteria in the form of invariants, the synthesis process can be framed as an iterative refinement problem.

The approach of framing system generation as an iterative refinement problem shares significant conceptual similarities with recent advancements in database engineering, particularly the work on Bespoke OLAP [11]. Wehrstein et al. propose synthesising workload-specific, ”one-size-fits-one” database engines, leveraging automated generation to tailor systems strictly to the operational requirements of a given workload rather than relying on generalised architectures. Both methodologies inherently recognise that modern distributed systems and data stores can be dynamically generated to meet exact constraints, moving away from rigid, one-size-fits-all implementations.

However, while Bespoke OLAP focuses on generating systems optimised for specific workloads, the synthesis experiment in this thesis applies generative techniques to functional correctness and distributed-system semantics. By integrating formal specifications into the LLM synthesis loop, the generation process is guided by invariant-based verification feedback rather than empirical performance metrics. This comparison demonstrates that automated synthesis can be constrained not only to build specialised, highly optimised systems, but also to construct correct-by-construction architectures that formally adhere to required consistency and operational guarantees.

6.3.1. Analysing the Synthesis Results

The empirical distributions from the synthesis experiment in Section 5.3 (Impact of Formal Feedback on System Synthesis) provide insight into how LLMs navigate formal constraints. The data indicates that formal specifications are a powerful tool, but they carry a computational burden.

Constraining the Search Space

The drastic reduction in the median number of iterations to convergence (from $\sim$95 in the Baseline to $\sim$40 in the Full Specification) agrees with the core hypothesis: invariant-based feedback successfully

shifts the LLM's behaviour from blind heuristics to targeted refinement. In the Baseline condition, the highly erratic code churn demonstrates that the LLM lacks the systemic boundaries necessary to understand distributed anomalies (such as state divergence). It seems to guess wildly, sometimes getting lucky, but frequently thrashing.

However, the Full Specification's high variance exposes a critical limitation of current LLMs. When faced with complex edge-case violations such as resolving asynchronous causality across replicas, the LLM occasionally struggles to align the strict TLA-style Quint feedback with its Python implementation. Rather than converging smoothly, the model can enter prolonged loops, struggling to translate formal notation into imperative logic, with long per-iteration computation times (15.8s). This is especially relevant for the selected `gemini-3-flash-preview` model, given its capabilities.

The Trade-off of Formalism

This translation struggle is quantified by the Tokens per Full Run metric. Because LLM API costs and contextual processing burden are strictly volume-based, the "context bloat" inherent in feeding massive strings of formal invariants back into the prompt drastically inflates the cost per iteration. Even though the Full Specification condition requires fewer median iterations, its large context window makes it significantly more expensive and volatile than natural-language guidance.

This observation isolates the Natural Specification condition as the current "semantic sweet spot." By guiding the LLM with human-readable, constraint-based rules, the framework avoids the extreme trial-and-error thrashing of the Baseline while simultaneously sidestepping the expensive context bloat of the Full Specification. It delivers the most consistent, predictable, and cost-effective convergence across all metrics.

Potential for Hybrid Translation Pipeline

These insights form a practical recommendation for the future of similar distributed systems synthesis. The optimal pipeline is likely a hybrid approach. Rigorous formal methods (such as Quint) remain necessary for evaluating generated code via trace execution, as they provide the unforgiving ground truth. However, the feedback provided back to the generative agent should not be raw invariants. Instead, the framework should include an intermediate translation layer that converts strict invariant failures into structured, natural language constraints before feeding them to the LLM. This hybrid architecture would leverage the safety guarantees of formal verification while optimising for the token efficiency and predictable convergence of natural language processing.

6.4. Broader Implications and Impact

The results indicate that formalising cloud service semantics as structured guarantees enables systematic reasoning about compatibility across heterogeneous systems. By representing systems as elements in a product lattice of guarantees, compatibility can be reduced to a well-defined ordering relation. This abstraction provides a clear and principled alternative to ad-hoc or empirical comparison methods commonly used in practice.

Furthermore, the trace-based evaluation demonstrates that the formal models are not merely theoretical constructs but can be grounded in concrete system behaviour. The complete alignment between model-generated traces and their execution against a real system suggests that the proposed specifications accurately capture the operational semantics of the modelled systems. This supports the central hypothesis that formal methods can bridge the gap between abstract system descriptions and real-world implementations.

At a higher level, the results suggest a shift from infrastructure-level portability toward semantic portability. While existing tools focus on deployment and configuration, this work highlights behavioural equivalence as the key factor in safe system migration.

6.4.1. Relation to Research Aims

Research Objectives

The first research objective was to develop a modular and extensible set of formal building blocks for capturing cloud service semantics. The results indicate that modelling systems along independent

guarantee dimensions is both expressive and practical. The use of partially ordered sets enables structured comparison of systems while remaining flexible enough to accommodate different service types.

The second objective concerned the automated synthesis of formal models from high-level configurations. The successful generation and verification of system models demonstrate that this approach is feasible and can significantly reduce the manual effort typically associated with formal specification. This supports the claim that formal methods can be made more accessible through automation.

The third objective was to define a compatibility framework based on invariants. The results show that invariant-based reasoning provides a precise mechanism for identifying semantic mismatches. Unlike traditional testing approaches, which rely on sampled executions, invariants capture global correctness properties that must hold across all possible behaviours.

Finally, the trace-based validation approach addresses the fourth objective by providing a link between formal models and real systems. The ability to replay traces and validate invariants in a concrete environment strengthens confidence in the correctness of the specifications and their applicability in practice.

Research Questions

The findings of this thesis directly address the core research questions outlined in Section 1.3:

RQ1 (Minimal set of guarantees)

The results confirm that modelling systems along independent guarantee dimensions (specifically delivery, ordering, and messaging models for queues, and consistency, idempotence, conditional writes, and replication for key-value stores) provides a minimal yet expressive baseline. Representing these dimensions as partially ordered sets (posets) successfully differentiated all evaluated real-world systems without requiring overly complex, implementation-specific state variables.

RQ2 (Comparing systems via formal specifications)

The pairwise compatibility experiments (Section 5.1) definitively answer this question. By framing compatibility as a strict subset of invariant preservation, the formal system mapped the exact boundaries of ”Semantic Lock-in” and identified bottleneck invariants, thereby predicting system incompatibilities without exhaustive empirical testing.

RQ3 (Deriving composed guarantees)

The composition compatibility results (Section 5.1.3) demonstrate that end-to-end architecture guarantees can indeed be systematically derived by evaluating the Cartesian product of individual components. Crucially, the experiments revealed that composed systems suffer from a multiplicative failure effect, where a semantic mismatch in either the database or messaging layer invalidates the entire migration pathway.

RQ4 (LLM synthesis correctness)

The trace-based validation and LLM synthesis experiments (Section 5.3) prove that formal specifications act as an exceptional guidance mechanism. Providing LLMs with strict, invariant-based feedback reduced the median number of iterations required to converge on a correct distributed system by approximately 58% compared to unguided baselines, confirming the significant potential of integrating formal verifiers with AI-driven code generation.

6.4.2. Comparison with Existing Work

Existing approaches to mitigating vendor lock-in primarily focus on standardisation, abstraction layers, or migration tooling. For example, frameworks for cloud migration often emphasise data portability and API compatibility, but do not account for differences in system semantics [7, 9]. As a result, subtle behavioural mismatches may remain undetected. In contrast, this work focuses explicitly on the semantic layer, modelling guarantees such as delivery, ordering, and consistency. This aligns with observations by Beslic et al. [8], who identify the lack of semantic interoperability as a key barrier to cloud portability. By providing a formal representation of these semantics, this thesis contributes a complementary approach that can be integrated with existing migration tools.

Additionally, while formal methods have been widely used in the verification of distributed systems, their application to cloud service interoperability remains limited. This work demonstrates that formal specification languages such as Quint can be applied effectively in this domain, providing both theoretical rigour and practical tooling support.

Furthermore, this thesis contributes to the rapidly evolving intersection of Large Language Models (LLMs) and formal verification, building upon concepts introduced in recent literature (see Section 2.2.3 (LLM-Assisted Formal Verification and Specification Synthesis)). While tools such as SAFE [44] and AUTOVERUS [45] are highly effective, they primarily focus on code-level functional verification and proof repair for specific programming languages. In contrast, this work moves these generative formal techniques to the architectural level, utilising LLMs to synthesise distributed systems guided by the composition of broad cloud service semantics.

Finally, the LLM-based synthesis loop explored in Section 4.4 (Evaluating the Impact of Formal Specifications on LLM-Based Synthesis of Distributed Systems) relies on trace-based invariant validation to provide a strict correctness guide. This directly aligns with recent methodologies that evaluate user-intent formalisation by symbolically testing specifications against input-output examples to mechanically measure correctness and completeness [43]. By integrating these formal feedback loops, this work extends the application of LLM-assisted verification to cloud-agnostic system design.

6.4.3. Implications for Cloud-Agnostic System Design

One of the key implications of this work is that it enables a more principled approach to designing cloud-agnostic systems. By expressing system requirements in terms of guarantees rather than specific implementations, developers can reason about the compatibility of different services before deployment.

This is directly relevant to the problem of vendor lock-in. Instead of adapting applications to provider-specific behaviour, it becomes possible to define the required guarantees and verify whether a target system satisfies them. In cases where no direct match exists, the formal model can highlight the precise nature of the mismatch, enabling informed design decisions or the implementation of compensating mechanisms.

Moreover, the framework supports the design of self-managed or portable systems that replicate the behaviour of managed cloud services. By using formal specifications as a reference, such systems can be validated against the expected semantics, reducing the risk of subtle correctness issues.

6.5. Limitations

Despite the promising results, several limitations must be acknowledged. These limitations relate to the expressiveness of the modelling approach, the completeness of the verification process, and the scope of the empirical evaluation.

Exclusion of Non-Functional Properties

First, the framework's expressiveness is constrained by the chosen dimensions of the guarantee. While delivery semantics, ordering guarantees, and consistency models capture many core aspects of cloud services, they do not provide a complete representation of system behaviour. In particular, non-functional properties such as latency, throughput, fault tolerance, and recovery mechanisms are not explicitly modelled. These aspects can significantly impact system correctness and user-perceived behaviour, especially in production environments. As a result, two systems that are deemed compatible under the current framework may still exhibit substantially different performance characteristics or failure behaviours. This limitation suggests that the current abstraction is most suitable for reasoning about functional correctness, but may need to be extended to incorporate additional dimensions for performance-sensitive or reliability-critical applications.

Completeness of the Invariant Set

Furthermore, the correctness guarantees provided by invariant checking depend heavily on the completeness and quality of the invariant set. Invariants define the properties considered essential for correctness, and any omission may lead to false positives, in which a system is classified as correct despite violating implicit or unmodelled requirements. This challenge is particularly relevant in the con-

text of the LLM-based synthesis experiment, where the model may overfit to specific feedback patterns, resulting in solutions that address individual violations without achieving general correctness. Furthermore, defining a comprehensive set of invariants is a non-trivial task that often requires deep domain knowledge. If the set of invariants is incomplete, the LLM may produce implementations that satisfy the given properties but are incorrect in untested scenarios. This introduces a degree of subjectivity into the verification process and may limit the reproducibility of results across different system types.

Theoretical Compatibility versus Practical Migration Realities

Furthermore, it is critical to state that while the compatibility matrices indicate mathematically safe migration pathways (denoted as full compatibility), this does not imply that a migration can be executed blindly with the expectation of flawless, out-of-the-box behaviour. The formal verification engine evaluates compatibility by preserving strict invariants, but real-world architectures often rely on implicit, system-specific behaviours that are not easily captured by formal logic.

Two specific architectural shifts highlight why a mathematically ”safe” migration might still require manual, application-level workarounds in reality:

- **Messaging Model Shifts**: The framework may flag a migration between fundamentally different messaging models (e.g., moving from a Point-to-Point architecture to a Pub/Sub architecture) as mathematically compatible if the underlying delivery and consistency invariants are preserved. However, in practice, a Pub/Sub topic cannot serve as a P2P queue without architectural intervention, such as manually configuring the target system to have only a single subscriber group.
- **Ordering Definition Differences**: A migration might move from a system with a stricter ordering guarantee to one with a looser guarantee (e.g., migrating from Global FIFO to partition-level ordering). If the source application’s formal invariants do not strictly enforce global ordering, the framework will accurately mark the migration as safe. Yet, if the application was implicitly relying on that strict global sequence to process interdependent events, the migration will still cause unpredictable runtime errors unless the application logic is retrofitted to handle out-of-order message delivery.

Therefore, while the formal framework identifies the theoretical safety boundaries of a migration, developers must still carefully evaluate whether the target system’s architectural model aligns with the practical realities of their application codebase.

State Observability Assumptions

Besides this, the trace-based validation approach relies on the ability to accurately reconstruct system transitions from observed state differences. While this approach proved effective in the evaluated setting, it assumes that the system state is sufficiently observable and that transitions can be unambiguously inferred. In more complex systems, state representations may be partial, distributed, or opaque, making it difficult to derive precise transition semantics. Additionally, systems with non-deterministic behaviour or concurrent interactions may produce state changes that cannot be easily mapped to a single logical operation. These challenges may reduce the reliability of trace reconstruction and limit the approach’s applicability to systems with well-defined, observable state transitions.

Breadth of Empirical Evaluation

The evaluation has so far been limited to a single concrete system implementation that combines two systems: message queues and key-value stores. Although the results demonstrate correctness and consistency in this specific case, they do not provide sufficient evidence of the framework’s general applicability. Different systems may exhibit alternative design choices, edge cases, or complexity that may be difficult to capture with the current evaluation method. Expanding the evaluation to include different and more complex cloud service systems would provide a more comprehensive assessment of the framework’s applicability to real-world systems, as cloud architectures are often more convoluted than the composition of the two chosen systems.

Scalability of Formal Verification

Finally, there are limitations related to the use of verification techniques within the framework. In principle, model checking provides significantly stronger guarantees than simulation, as it systematically

explores the state space and can establish correctness across all possible executions. This makes it the preferred approach for formal verification of distributed systems. However, in the context of this work, the complexity of the modelled system resulted in severe scalability challenges. The state space grows rapidly due to the combination of multiple replicas, operations, and possible interleavings, leading to state explosion. In practice, attempts to perform full model checking did not complete within a reasonable time frame; a single run exceeded three days without terminating. This indicates that, while theoretically desirable, exhaustive verification is not always feasible for realistic system configurations.

As a result, simulation-based analysis was adopted as a practical alternative. Simulation allows exploration of representative execution traces and validation of invariants under specific scenarios. This approach is significantly more tractable and can provide useful insights into system behaviour within a reasonable time frame. However, it does not offer the same level of completeness as model checking. The explored traces represent only a subset of all possible executions, and therefore, the absence of invariant violations cannot be interpreted as a formal guarantee of correctness. Instead, the results should be understood as empirical evidence that the system behaves correctly under the tested conditions.

This trade-off between completeness and scalability highlights a fundamental challenge in applying formal methods to complex distributed systems. While model checking offers strong theoretical guarantees, its practical applicability is often limited by computational constraints. Conversely, simulation provides scalability at the cost of weaker guarantees. Addressing this tension remains an important direction for future work, for example through abstraction techniques, state-space reduction, or hybrid approaches that combine model checking with targeted simulation.

7

Conclusion

The transition to cloud-native architectures has provided organisations with unprecedented scalability, but it has also introduced the persistent technical and strategic risk of vendor lock-in. While existing tools address deployment and infrastructure portability, this thesis tackles the underlying, often overlooked dimension of this problem: the semantic gap. Even when cloud services share identical APIs, subtle variations in their delivery, ordering, and consistency guarantees can lead to unpredictable and critical application failures during migration.

To bridge this gap, this research proposed and evaluated a formal framework for automating cloud service compatibility analysis. By defining the operational semantics of message queues and key-value stores along structured dimensions of guarantees, systems were successfully abstracted into rigorous mathematical spaces. Leveraging the Quint specification language, the developed generative pipeline demonstrated that formal models and correctness invariants can be automatically synthesised from high-level service descriptions.

The evaluation yielded several insights into the nature of distributed cloud system interaction:

- **The Asymmetry of Migration**: The pairwise compatibility experiments showed that safe cross-cloud migration in this context is fundamentally directional. Systems can generally migrate to targets with stronger semantics safely, but downgrading guarantees structurally breaks system correctness.
- **Quantifying Semantic Lock-in**: The framework successfully isolated the specific operational guarantees responsible for migration failures. Bottleneck invariants, most notably exactly-once processing and strict FIFO ordering, were identified as the primary drivers of ”Semantic Lock-in,” restricting architectural mobility and permanently anchoring systems to specific providers.
- **Bridging Theory and Implementation**: The trace-based validation methodology successfully connected abstract models to real-world execution. By parsing formal traces and replaying them against a concrete Redis instance, this work verified that the generated specifications accurately reflect and enforce real system behaviours under the given test conditions.
- **Guiding Automated Synthesis**: Preliminary explorations into LLM-based system generation revealed that formal invariants provide an excellent guidance mechanism. When integrated into an automated feedback loop, formal methods offer the precise boundaries required to refine and verify AI-generated distributed architectures.

Ultimately, this thesis demonstrates that escaping vendor lock-in requires moving beyond surface-level API mapping to achieve true semantic portability. By transforming informal documentation into machine-checkable guarantees, this work shifts system comparison from ad-hoc empirical testing to a principled, verifiable science. The proposed framework provides engineers with the foundation necessary to reason about interoperability, navigate complex cross-cloud migrations safely, and architect genuinely cloud-agnostic systems.

8

Recommendations and Future Work

The results of this thesis demonstrate the feasibility of modelling cloud service semantics using formal methods and leveraging these models for compatibility analysis and system synthesis. At the same time, several limitations and open challenges suggest clear directions for future work. This section outlines concrete recommendations to extend the current framework, improve its applicability, and explore additional research opportunities.

8.1. Extending the Set of Guarantee Dimensions

A key next step is to expand the set of modelled guarantees beyond delivery, ordering, and consistency. While these dimensions capture core functional properties, they do not fully represent the behaviour of real-world cloud systems. In particular, incorporating non-functional aspects such as latency, throughput, fault tolerance, and recovery mechanisms would significantly enhance the framework's expressiveness.

Such an extension would allow the framework to reason not only about correctness, but also about performance and reliability trade-offs. This is especially relevant in practical migration scenarios, where systems must meet both functional and operational requirements. However, modelling these aspects introduces additional complexity, as they often involve probabilistic or time-dependent behaviour. Future work could explore hybrid modelling approaches that combine formal specifications with statistical or empirical models.

8.2. Improving Invariant Generation and Coverage

The effectiveness of the framework depends strongly on the quality and completeness of the generated invariants. A promising direction is to develop systematic methods for generating invariants, thereby reducing reliance on manual specification. For example, invariants could be derived from high-level system requirements or synthesised using automated reasoning techniques.

In addition, future work could investigate methods for validating the completeness of invariant sets. This may involve techniques such as mutation testing, where small changes are introduced into the system to assess whether the invariants detect incorrect behaviour. Improving invariant coverage would strengthen the framework's correctness guarantees and reduce the risk of false positives.

8.3. Scaling and Optimising Formal Verification

Given the computational constraints encountered during exhaustive model checking, future work should focus on optimising the verification pipeline and scaling the underlying hardware to make full model checking tractable for complex compositions. Rather than falling back on simulation, which inherently sacrifices mathematical certainty, efforts must be directed toward mitigating state-space explosion.

At the algorithmic and system levels, future research could implement state-space reduction techniques

directly within the Quint specifications. Techniques such as partial order reduction and abstraction refinement could significantly decrease the number of reachable states the model checker must evaluate.

Furthermore, the hardware and execution infrastructure supporting the verification engine can be significantly scaled. Model checking composed distributed architectures is computationally heavy but should be parallelisable in specific contexts. Future iterations of this framework could leverage distributed model checking by deploying solvers across high-performance cloud computing clusters, rather than the current experimental setup. By removing local machine bottlenecks and distributing the state exploration across multiple compute nodes, the framework could perhaps perform model checking within a reasonable time frame.

8.4. Expanding Empirical Evaluation

The current evaluation focuses on a single system implementation, which limits the generalizability of the results. A natural next step is to apply the framework to a broader range of systems, including different key-value stores, message queues, and other cloud services. This would provide stronger evidence of the approach's applicability across diverse domains, as modern architectures are often more intricate than the composed systems.

In addition, future experiments could consider more complex workloads, larger state spaces, and different failure scenarios. Evaluating the framework under these conditions would help identify potential weaknesses and refine the modelling approach.

8.5. Enhancing LLM-Based Synthesis with Formal Feedback

While this thesis established a foundational understanding of how formal feedback affects system generation using a single model (`gemini-3-flash-preview`) and three specific prompting conditions, a critical next step is to conduct a multidimensional analysis across more diverse configurations and LLM architectures.

Another direction is to explore tighter integration between the synthesis and verification components. Instead of treating verification as an external step, it may be possible to embed formal constraints directly into the generation process. This could lead to more efficient convergence and reduce the number of required iterations.

Furthermore, future work could evaluate the generality of this approach across different system types and specifications, assessing whether formal guidance consistently improves synthesis outcomes. Finally, varying the architectural complexity of the target system, such as guiding an LLM to synthesise systems that require more complex conflict resolution or distributed consensus protocols like Raft, could provide necessary insights into the scalability limits of invariant-guided program synthesis.

8.6. Towards Practical Tooling for Cloud-Agnostic Design

To increase the practical impact of this work, the framework could be developed into a user-facing tool for cloud system design and migration. Such a tool would allow developers to specify required guarantees, automatically generate formal models, and evaluate compatibility with target systems.

Integrating this approach with existing cloud management tools could provide a more comprehensive solution for avoiding vendor lock-in. For example, compatibility analysis could be combined with deployment automation to support end-to-end migration workflows.

Additionally, improving usability and abstraction levels will be important for adoption. This includes designing intuitive configuration formats, providing meaningful feedback to users, and reducing the need for expertise in formal methods.

References

[1] Alan Megargel, Venky Shankararaman, and David K Walker. “Migrating from monoliths to cloud-based microservices: A banking industry example”. In: *Software engineering in the era of cloud computing*. Springer, 2020, pp. 85–108.

[2] Espen Tønnessen Nordli et al. “Migrating monoliths to cloud-native microservices for customizable SaaS”. In: *Information and Software Technology* 160 (2023), p. 107230.

[3] Guo Fu, Yanfeng Zhang, and Ge Yu. “A Fair Comparison of Message Queuing Systems”. en. In: *IEEE Access* 9 (2021), pp. 421–432. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3046503. URL: https://ieeexplore.ieee.org/document/9303425/ (visited on 03/23/2026).

[4] Adity Gupta et al. “NoSQL databases: Critical analysis and comparison”. en. In: *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*. Gurgaon: IEEE, Oct. 2017, pp. 293–299. ISBN: 978-1-5386-0627-8. DOI: 10.1109/IC3TSN.2017.8284494. URL: http://ieeexplore.ieee.org/document/8284494/ (visited on 03/23/2026).

[5] Kevin Xiaoguo Zhu and Zach Zhizhong Zhou. “Research note—Lock-in strategy in software competition: Open-source software vs. proprietary software”. In: *Information systems research* 23.2 (2012), pp. 536–545.

[6] Mahdi Fahmideh Gholami et al. “Cloud migration process—A survey, evaluation framework, and open challenges”. In: *Journal of systems and software* 120 (2016), pp. 31–69.

[7] Justice Opara-Martins, Reza Sahandi, and Feng Tian. “A Holistic Decision Framework to Avoid Vendor Lock-in for Cloud SaaS Migration”. en. In: *Computer and Information Science* 10.3 (July 2017), p. 29. ISSN: 1913-8997, 1913-8989. DOI: 10.5539/cis.v10n3p29. URL: http://www.ccsenet.org/journal/index.php/cis/article/view/69798 (visited on 11/11/2025).

[8] Alexandre Beslic et al. “Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms.” In: *MDHPCL@ MoDELS*. 2013, pp. 5–14.

[9] Ruhul Amin, Siddhartha Vadlamudi, and Md Mahbubur Rahaman. “Opportunities and challenges of data migration in cloud”. In: *Engineering International* 9.1 (2021), pp. 41–50.

[10] Mohammed Shuaib et al. “Why adopting cloud is still a challenge?—A review on issues and challenges for cloud migration in organizations”. In: *Ambient Communications and Computer Systems: RACCCS-2018* (2019), pp. 387–399.

[11] Johannes Wehrstein et al. “Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines”. In: *arXiv preprint arXiv:2603.02001* (2026).

[12] Benjamin Satzger et al. “Winds of change: From vendor lock-in to the meta cloud”. In: *IEEE internet computing* 17.1 (2013), pp. 69–73.

[13] Eric Anderson et al. “What Consistency Does Your {Key-Value} Store Actually Provide?” In: *Sixth Workshop on Hot Topics in System Dependability (HotDep 10)*. 2010.

[14] Michael Howard. “Terraform–Automating Infrastructure as a Service”. In: *arXiv preprint arXiv:2205.10676* (2022).

[15] Tobias Binz et al. “OpenTOSCA–a runtime for TOSCA-based cloud applications”. In: *International Conference on Service-Oriented Computing*. Springer. 2013, pp. 692–695.

[16] Uwe Breitenbücher et al. “Combining declarative and imperative cloud application provisioning based on TOSCA”. In: *2014 IEEE international conference on cloud engineering*. IEEE. 2014, pp. 87–96.

[17] Informal Systems. *Quint: A modern specification language based on the temporal logic of actions (TLA)*. Official Documentation. 2024. URL: https://quint-lang.org/.

[18] Richard A. Kemmerer. “Testing formal specifications to detect design errors”. In: *IEEE transactions on software engineering* 1 (2006), pp. 32–43.

[19] Robert M Hierons et al. “Using formal specifications to support testing”. In: *ACM Computing Surveys (CSUR)* 41.2 (2009), pp. 1–76.

[20] Apache Software Foundation. *Apache ActiveMQ Documentation*. Accessed: 2026-05-26. 2026. URL: https://activemq.apache.org/documentation.

[21] Apache Software Foundation. *Apache Kafka Documentation*. Accessed: 2026-05-26. 2026. URL: https://kafka.apache.org/documentation/.

[22] Apache Software Foundation. *Apache Pulsar Documentation*. Accessed: 2026-05-26. 2026. URL: https://pulsar.apache.org/docs/.

[23] Amazon Web Services. *Amazon Simple Queue Service Developer Guide*. Accessed: 2026-05-26. 2026. URL: https://docs.aws.amazon.com/sqs/.

[24] RabbitMQ. *RabbitMQ Documentation*. Accessed: 2026-05-26. 2026. URL: https://www.rabbitmq.com/docs.

[25] Redis Ltd. *Redis Data Types Documentation*. Accessed: 2026-05-26. 2026. URL: https://redis.io/docs/latest/develop/data-types/.

[26] Apache Software Foundation. *Apache RocketMQ Documentation*. Accessed: 2026-05-26. 2026. URL: https://rocketmq.apache.org/docs/.

[27] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[28] Apache Software Foundation. *Apache Cassandra Documentation*. Accessed: 2026-05-26. 2026. URL: https://cassandra.apache.org/doc/latest/.

[29] Microsoft. *Azure Cosmos DB Documentation*. Accessed: 2026-05-26. 2026. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/.

[30] Amazon Web Services. *Amazon DynamoDB Developer Guide*. Accessed: 2026-05-26. 2026. URL: https://docs.aws.amazon.com/dynamodb/.

[31] Memcached Contributors. *Memcached Wiki Documentation*. Accessed: 2026-05-26. 2026. URL: https://github.com/memcached/memcached/wiki.

[32] MongoDB Inc. *MongoDB Documentation*. Accessed: 2026-05-26. 2026. URL: https://www.mongodb.com/docs/.

[33] Redis Ltd. *Redis Documentation*. Accessed: 2026-05-26. 2026. URL: https://redis.io/docs/.

[34] Mukesh Singhal and Ajay Kshemkalyani. “An efficient implementation of vector clocks”. In: *Information Processing Letters* 43.1 (1992), pp. 47–52.

[35] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. “Conflict-free replicated data types (CRDTs)”. In: *arXiv preprint arXiv:1805.06358* (2018).

[36] Travis Hance et al. “Finding invariants of distributed systems: It’s a small (enough) world after all”. In: *18th USENIX symposium on networked systems design and implementation (NSDI 21)*. 2021, pp. 115–131.

[37] Maryam Dabaghchian et al. “Consistency-aware scheduling for weakly consistent programs”. In: *ACM SIGSOFT Software Engineering Notes* 42.4 (2018), pp. 1–5.

[38] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. “Trace aware random testing for distributed systems”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–29.

[39] Burcu Kulahcioglu Ozkan et al. “Randomized testing of distributed systems with probabilistic guarantees”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–28.

[40] Burcu Kulahcioglu Ozkan. “Verifying Weakly Consistent Transactional Programs Using Symbolic Execution”. In: *International Conference on Networked Systems*. Springer. 2020, pp. 261–278.

[41] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. “TLA+ model checking made symbolic”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

[42] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[43] Haoran Ding, Zhaoguo Wang, and Haibo Chen. “FM-Agent: Scaling Formal Methods to Large Systems via LLM-Based Hoare-Style Reasoning”. In: *arXiv preprint arXiv:2604.11556* (2026).

[44] Tianyu Chen et al. “Automated proof generation for rust code via self-evolution”. In: *arXiv preprint arXiv:2410.15756* (2024).

[45] Chenyuan Yang et al. “Autoverus: Automated proof generation for rust code”. In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA2 (2025), pp. 3454–3482.

[46] Shuvendu K Lahirie. “Evaluating llm-driven user-intent formalization for verification-aware languages”. In: *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2024, pp. 142–147.

[47] Leslie Lamport. *Specifying systems*. Vol. 388. Addison-Wesley Boston, 2002.

[48] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model checking TLA+ specifications”. In: *Advanced research working conference on correct hardware design and verification methods*. Springer. 1999, pp. 54–66.

[49] Yaml. - *Yaml Ain’t markup language*. URL: `https://yaml.org/`.

[50] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. “TLA+ model checking made symbolic”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

A

Generated System Code Example

As described in Section 3.1.3 (System Composition), this appendix provides an excerpt of an automatically generated system specification written in the Quint language. The code illustrates how the framework pieces together independent components into a single, unified state machine. Note that some parts of the code have been omitted to save space. It defines the initialisation actions (init), composite transitions (e.g., compositeEnqueue and compositeDeliver), and an overarching step action that uses non-determinism (nondet) to simulate random, unpredictable system interactions. Finally, it aggregates all base invariants, composition-specific constraints, and generated invariants into a single SystemCorrect property, which serves as the primary target for simulation and model checking.

```
module System {
    {*CURRENTLY HIDDEN: imports and helper functions*}

    // Unified Init
    action init = all {
        initKV,
        initQueue,
        initComposition,
        nextMessageId' = 1,
        nextTimestamp' = 1,
    }

    action compositeSync(r1: Replica, r2: Replica): bool = all {
        sync(r1, r2),
        delivered' = delivered,
        history' = history,
        inflight' = inflight,
        nextMessageId' = nextMessageId,
        nextTimestamp' = nextTimestamp,
        processed' = processed,
        queue' = queue
    }
    action compositePublish(m: Message): bool = all {
        publish(m),
        kv' = kv,
        lastWrite' = lastWrite,
        nextMessageId' = nextMessageId + 1,
        nextTimestamp' = nextTimestamp + 1,
        processed' = processed,
        visibleReplicas' = visibleReplicas
    }
    action compositeDeliver(m: Message, deliveryChoice: int): bool = all {
        deliver(m, deliveryChoice),
        kv' = kv,
        lastWrite' = lastWrite,
        nextMessageId' = nextMessageId,
        nextTimestamp' = nextTimestamp,
        processed' = processed,
        visibleReplicas' = visibleReplicas
```

```
    }
    action compositeDelete(m: Message): bool = all {
        delete(m),
        kv' = kv,
        lastWrite' = lastWrite,
        nextMessageId' = nextMessageId,
        nextTimestamp' = nextTimestamp,
        processed' = processed,
        visibleReplicas' = visibleReplicas
    }
    action compositeProcess(recipient: int, m: Message, r: Replica, mode: int): bool = all {
        process(recipient, m, r, mode),
        nextMessageId' = nextMessageId,
        nextTimestamp' = nextTimestamp
    }

    // Unified Step
    action step = {
        {*CURRENTLY HIDDEN: Nondeterministic choices*}

        val input: Message = {id: nextMessageId, ts: nextTimestamp, key: k, value: v, client:
             client, recipients: recipients}

        any {
            if (queue.empty()) compositePublish(input) // Force enqueue if queue is empty to
                ensure progress and avoid deadlock
            else any {
                compositeSync(replica, dst),
                compositePublish(input),
                match getHeadList(deliveryTarget) {
                    | Some(m) => compositeDeliver(m, deliveryMode)
                    | None => AllUnchanged
                },
                match getHead(queue) {
                    | Some(m) => compositeDelete(m)
                    | None => AllUnchanged
                },
                match getHeadList(processTarget) {
                    | Some(m) => compositeProcess(m._1, m._2, replica, deliveryMode)
                    | None => AllUnchanged
                }
            }
        }
    }

    // Base Invariants
    val base_invariants = all {
        kv_invariants,
        queue_invariants,
        composition_invariants,
    }

    // Full System Invariants
    val SystemCorrect = base_invariants and generated_invariants
}
```

B

Generated Invariants Code Example

As described in Section 3.2.3 (Invariant Composition), this appendix presents a concrete example of the invariants automatically synthesised by the generative pipeline. The Quint module shown here translates the high-level semantic guarantees of the source system into precise mathematical constraints. Examples include delivery semantics such as `exactly_once` and `at_most_once`, as well as compositional constraints such as ensuring that all entries maintain LWW conflict resolution (`lww_conflict_resolution`). These individual properties are bundled into the `generated_invariants` definition, establishing the strict correctness boundaries required during trace-based validation and cross-cloud compatibility testing.

```
module GeneratedInvariants {
    {*CURRENTLY HIDDEN: imports*}

    val processed_from_queue = convertToSet(processed).forall(p => history.exists(m => m.id
        == p._2.id))

    val no_double_processing = history.forall(m =>
        m.recipients.forall(r =>
            size(processed.indices().filter(i => processed[i] == (r, m))) <= 1
        )
    )

    val no_processing_after_ack = // No processing after acknowledgment (removed from
        inflight)
    inflight.forall(p =>
        not(convertToSet(processed).contains(p))
    )

    val causality = // For strong consistency, every replica should reflect every processed
        message
    convertToSet(processed).forall(p =>
        kv.values().forall(replicaMap =>
            replicaMap.has(p._2.key) and
            replicaMap.get(p._2.key)._2 >= p._2.ts
        )
    )

    val at_most_once = processed.length() == size(convertToSet(processed))

    val crash_safety = // Crashes should not occur unless specified in the system model
      // Every recipient of every message in history must be either in queue, inflight,
          processed, or delivered
      history.forall(m =>
        m.recipients.forall(r =>
          m.in(queue) or (r, m).in(inflight) or convertToSet(processed).contains((r, m)) or (
              r, m).in(delivered)
        )
      )

```

```
    val eventual_consistency = // If replicas are synced and queue is empty, they should reflect the latest writes
    (queue.empty() and inflight.empty() and kv.values().forall(v1 => kv.values().forall(v2 => v1 == v2))) implies
        kv.values().forall(replicaMap =>
            replicaMap.keys().forall(k =>
                val maxTs = convertToSet(processed).filter(p => p._2.key == k).map(p => p._2.ts).fold(-1, (a, b) => max(a, b))
                replicaMap.get(k)._2 == maxTs
            )
        )

    val lww_conflict_resolution = // For strong consistency, every replica should always reflect the latest write (highest timestamp)
    kv.values().forall(replicaMap =>
        processed.indices().forall(i =>
            val p = processed[i]
            val maxTs = convertToSet(processed).filter(p2 => p2._2.key == p._2.key).map(p2 => p2._2.ts).fold(-1, (a, b) => max(a, b))
            replicaMap.getOrElse(p._2.key, (0, -1))._2 == maxTs
        )
    )

    val replica_sync = // If a key exists in any replica, it originated from some processed message
    kv.keys().forall(r =>
        kv.get(r).keys().forall(k =>
            convertToSet(processed).exists(p => p._2.key == k and p._2.value == kv.get(r).get(k)._1)
        )
    )

    val generated_invariants = all {
        processed_from_queue,
        no_double_processing,
        no_processing_after_ack,
        causality,
        at_most_once,
        crash_safety,
        eventual_consistency,
        lww_conflict_resolution,
        replica_sync
    }
}
```

C

LLM-Based Synthesis Experiment Prompts

As described in Section 4.4.1 (Experimental Setup), the following prompt would be consistent across all runs. This prompt defines the context for the agent, specifying the conditions and constraints that should hold in its responses:

Main System Prompt

You are an expert distributed systems engineer.
Your task is to implement a key-value store that satisfies a set of correctness invariants.
You will be given:

1. A Python implementation of a KV store (which may be incorrect)
2. Feedback from a verification system (trace parsing + invariants)

Your job:

- Modify the implementation to fix the errors
- Preserve correct behaviour
- Ensure all invariants pass

STRICT RULES:

- Only output the full updated ‘kv_store.py‘ file
- Do not include explanations
- Do not change function signatures
- Do not add external dependencies
- Keep the implementation deterministic
- Do not attempt to read invariants.py; treat check_invariants as a black box.

The KV store must support multiple replicas and behave according to the observed trace semantics.
Focus on correctness over performance.

Iteration Prompt - Full System Specification

Implement a distributed key-value store with the following properties:
PROPERTIES:
1. Multiple replicas maintain key-value maps
2. Writes insert/update values at a replica
3. Deletes remove keys from a replica
4. Sync operations propagate values between replicas
5. Replicas should not diverge (no conflicting values for the same key)
6. All values must originate from a write in history
7. Sync operations must only add data (never remove)
8. The latest write must be visible

INTERFACE: See Listing 4.4.1
You have been given the full quint specification of the key-value store system used to create the trace in ‘quint/‘. Use this to inform your implementation.
Return only the full kv_store.py file.

Iteration Prompt - Natural Language System Specification

Implement a distributed key-value store with the following properties:
PROPERTIES:
1. Multiple replicas maintain key-value maps
2. Writes insert/update values at a replica
3. Deletes remove keys from a replica
4. Sync operations propagate values between replicas
5. Replicas should not diverge (no conflicting values for the same key)
6. All values must originate from a write in history
7. Sync operations must only add data (never remove)
8. The latest write must be visible

INTERFACE: See Listing 4.4.1
Return only the full kv_store.py file.

Iteration Prompt - No System Specification

Implement a distributed key-value store with multiple replicas.
Each replica should store key-value pairs and support writes, deletes, and state retrieval.
INTERFACE: See Listing 4.4.1
Return only the full kv_store.py file.

Iteration Prompt after Failure

The current implementation failed verification.
FEEDBACK: feedback
CURRENT IMPLEMENTATION: code
Fix the implementation so that it satisfies all invariants.
Return only the full updated kv_store.py file.

D

List of Invariants

Queue Invariants

These invariants focus on the delivery and ordering guarantees of the message queue systems.

- `processed_from_queue`: Ensures that every message processed by a recipient actually originated from the system's message history.
- `no_double_processing`: (Requires exactly_once delivery) Ensures that no message is processed more than once by the same recipient.
- `no_processing_after_ack`: (Requires exactly_once delivery) Ensures that a message is never processed after it has been acknowledged (removed from the inflight set).
- `fifo_ordering`: (Requires first_in_first_out ordering) Ensures that messages for the same recipient are processed in the exact order they were sent.
- `at_most_once`: Ensures that every message is processed at most once by any given recipient.

Key-Value Store Invariants

These invariants ensure data integrity and consistency in key-value storage systems.

- `causality`: (Requires strong consistency) Ensures that every replica in the system reflects the update of every message that has been marked as processed.
- `eventual_consistency`: (Requires eventual consistency) Ensures that once the queue is empty and all replicas have synced, they all reflect the latest write for every key.
- `lww_conflict_resolution`: Ensures that replicas resolve conflicts by always reflecting the value with the highest timestamp (Last-Write-Wins).
- `idempotent_writes`: Ensures that even if a message is processed multiple times, its value remains consistent in the replicas.
- `replica_sync`: Ensures that any data present in a replica can be traced back to a successfully processed message.
- `no_conflicts`: (Python-defined) Ensures that a single key does not have different values across replicas at the same logical timestamp.
- `kv_backed_by_history`: (Python-defined) Verifies that all values stored in the KV system originated from documented writes in the history.

Composition & Hybrid Invariants

These invariants cover the interaction between the Queue and KV systems in a migration or hybrid setup.

- `all_kv_from_queue`: Ensures that every entry in the KV store corresponds to a message that was successfully processed from the queue.

- `exactly_once`: A system-wide check ensuring that every message is eventually processed by all intended recipients and then removed from the queuing infrastructure.
- `no_reordering_strong_consistency`: Ensures that the combination of strong KV consistency and FIFO queueing results in a KV state that perfectly matches the latest processed messages.
- `crash_safety`: Ensures no messages are lost during system failures by verifying every message is either in the queue, currently being processed (inflight), or already delivered.

Operational Invariants

Additional invariants used primarily during trace parsing and live system experimentation.

- `sync_only_adds`: Ensures that replica synchronisation operations only add new data or update timestamps, never losing existing keys or regressing to older states.
- `last_write_visible`: A baseline check to see if the most recent write performed by a client is immediately visible in the system state.

E

Full Queue Network Graph

See Section 5.1.1 (Message Queue Compatibility)

Figure E.1: Network graph of Queue compatibility.

F

Declaration of use of Generative AI

During the preparation of this master's thesis, generative AI tools such as Google's Gemini were utilised to support various stages of the research and writing process. Specifically, besides the use outlined in Section 4.4 (Evaluating the Impact of Formal Specifications on LLM-Based Synthesis of Distributed Systems), generative AI was employed for the following purposes:

- Code Assistance: AI tools were used to assist with scripting, debugging, and writing boilerplate code for the automated generation pipeline, the trace-parsing scripts, and the Quint specification models.
- Brainstorming: Generative AI served as an exploratory tool to help brainstorm structural approaches to formalising cloud system semantics and to outline potential evaluation methodologies.
- Writing Quality Enhancement: AI was utilised to refine the academic tone, improve sentence structure, and enhance the manuscript's clarity and readability.

All AI-generated outputs were thoroughly reviewed, critically evaluated and manually edited to ensure factual accuracy, logical coherence, and adherence to academic standards. I take full responsibility for the final content, original ideas, and conclusions presented in this work.