

Fuzzing Microservices in Face of Intrinsic Uncertainties

MAN ZHANG, Beihang University, China

TAO YUE*, Beihang University, China

ANDREA ARCURI, Kristiania University of Applied Sciences and Oslo Metropolitan University, Norway

The widespread adoption of microservices has fundamentally transformed how modern software systems are designed, deployed, operated and maintained. However, well-known microservice properties (e.g., dynamic scalability and decentralized control) introduce inherent and multi-dimensional uncertainties. These uncertainties span across inter-service interactions, runtime environments, and internal service logic, which manifest as nondeterministic behaviors, performance fluctuations, and unpredictable fault propagation. Existing approaches do not have sufficient support in capturing such uncertainties and their propagation in industrial microservice systems, and these approaches mostly focus on single-service testing. In this paper, we argue for a novel paradigm: “uncertainty-driven” and “system-level” microservice testing. We outline key research challenges, including the modeling and injection of uncertainties and their propagation, causal inference for fault localization, and multi-dimensional analyses and assessment of uncertainties and their impact on system quality. We propose an architecture for continuous uncertainty-driven and system-level microservice fuzzing, which integrates service virtualization, uncertainty simulation, adaptive test generation and optimization, and illustrate it with an e-commerce example we developed. Our goal is to inspire the development of scalable and automated system-level testing methods that improve the dependability and resilience of industrial microservice systems, with the explicit consideration of uncertainties and their propagation.

Additional Key Words and Phrases: Fuzzing, System-level Testing, Uncertainty, API, REST, and Microservices

ACM Reference Format:

Man Zhang, Tao Yue, and Andrea Arcuri. 2026. Fuzzing Microservices in Face of Intrinsic Uncertainties . 1, 1 (April 2026), 29 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Microservices architecture has emerged as a foundational model for building advanced technologies (e.g., cloud and edge computing) and modern applications (e.g., industrial services), driven by its inherent technical strengths, such as service decoupling (enabling independent deployment), fault isolation (preventing cascading failures), elastic scalability (adapting to dynamic workloads), and polyglot persistence (supporting heterogeneous data stores) [59, 63]. For example, the microservices architecture of Alibaba (a Fortune 500 corporation) utilizes a modular microservices approach to enable the dynamic integration of multi-source data and to support real-time decision-making in areas like personalized pricing and fraud detection [52]. Similarly, MindSphere of Siemens (a Fortune 500 corporation) is an industrial IoT

*Corresponding author

Authors’ Contact Information: [Man Zhang](mailto:manzhang@buaa.edu.cn), manzhang@buaa.edu.cn, Beihang University, Beijing, , China; [Tao Yue](mailto:yuetao@buaa.edu.cn), Beihang University, Beijing, China, yuetao@buaa.edu.cn; [Andrea Arcuri](mailto:andrea.arcuri@kristiania.no), andrea.arcuri@kristiania.no, Kristiania University of Applied Sciences and Oslo Metropolitan University, Oslo, Norway.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

Manuscript submitted to ACM

platform that leverages microservices to orchestrate heterogeneous edge devices and cloud-based analytics, ensuring fault-tolerant data processing in mission-critical manufacturing workflows [43]. Meituan (a Fortune 500 corporation) developed online and on-demand delivery platforms with microservices, offering their services to more than 600 million users [86, 88, 89]. According to recent analytics, Edge Delta estimates that 94% of companies worldwide rely on cloud computing, with related expenditures reaching into the hundreds of billions of dollars [71].

As the adoption of microservices continues to grow across industries, ensuring the dependability of microservice systems has become critical. Industrial microservices are often characterized by complex business logic, further compounded by dynamic scalability, heterogeneous architectures, and distributed topologies. These characteristics introduce multi-dimensional uncertainties that affect both inter-service interactions (e.g., chains of API calls, message queues) and internal service implementations (e.g., non-deterministic algorithms, resource contention). Such uncertainties can lead to service anomalies (e.g., timeout cascades), performance degradation (e.g., latency spikes), and even cascading failures (e.g., database connection pool exhaustion), significantly jeopardizing system’s dependability.

Therefore, it is essential to establish a comprehensive testing framework that can explicitly incorporate various types of uncertainties inherent in microservice architectures and their operating environments. By explicitly considering these uncertainties and their propagation during testing, the testing framework can simulate real-world conditions and stress-test the system, which enables organizations to proactively identify potential points of failures, mitigate risks, and optimize system resilience. Otherwise, critical issues such as service downtime, slowdowns, or cascading failures might go undetected, leading to compromised quality and unreliable service delivery in operating environments. In this paper, we propose a conceptual architecture that defines the required components and key workflows to facilitate continuous uncertainty-driven and system-level microservice fuzzing. The proposed framework has not yet been fully implemented and it is intended to serve as a foundation for future development and empirical evaluation.

Structure: Section 2 discusses the key challenges to realize uncertainty-driven microservice testing; Section 3 discusses the state of art of microservices testing and uncertainty-aware software engineering, followed by our vision towards an uncertainty-driven microservice testing (Section 4); We illustrate our framework with an e-commerce example in Section 5 and provide discussions in Section 6; We conclude the paper in Section 7.

2 Challenges in Uncertainty-driven Microservice Fuzzing

The literature on microservice testing [32] predominantly focuses on deterministic scenarios (e.g., single-service API fuzzing) and lacks systematic ways to guide industrial practices in mitigating hidden risks arising from complex interactions, and inadequately addresses uncertainties in microservices (e.g., non-deterministic inputs, dynamic propagation paths). The demand for revolutionizing microservice testing has dramatically increased, especially when the widespread adoption of microservices in mission-critical domains such as financial transaction platforms (e.g., AliPay and Apple Pay) and real-time healthcare systems (e.g., patient monitoring systems in the Intensive Care Units).

In the rest of this section, we detail three challenges faced by the existing approaches in microservice testing, focusing on multi-dimensional uncertainty sources (Section 2.1), dynamic uncertainty propagation (Section 2.2), and combinatorial explosion of microservice uncertainties (Section 2.3). In Section 2.4, we also differentiate uncertainties in microservices with those in traditional distributed systems.

2.1 Multi-dimensional microservice uncertainties

The architecture of microservices is inherently distributed, hence microservices face multi-dimensional uncertainties and complex interactions among them. As illustrated in Figure 1, in industrial application contexts, such uncertainties may

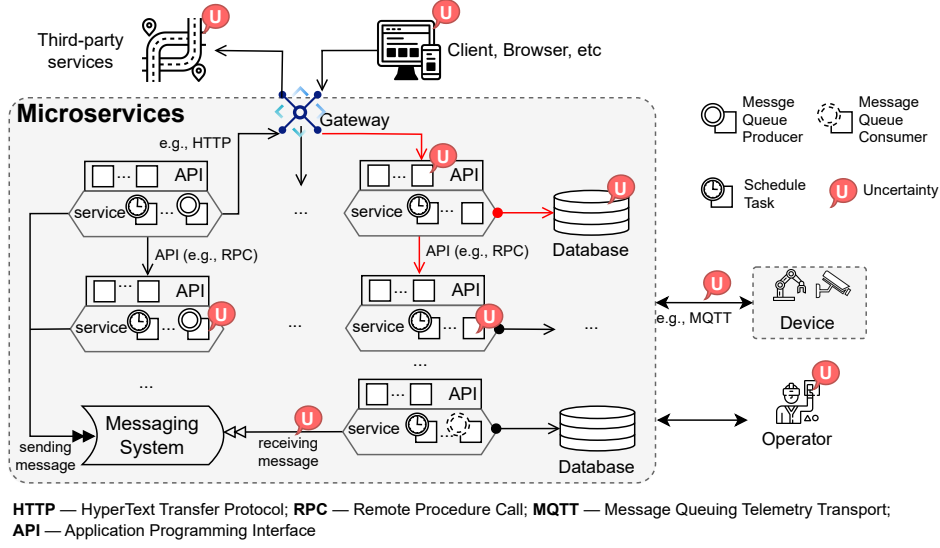


Fig. 1. Illustrating Potential Occurrences of Uncertainties in Microservices. Uncertainties may occur in third-party services, client sides, APIs, database, message queuing, scheduling tasks, etc.

originate from network fluctuations (e.g., latency, jitter), heterogeneous environments (e.g., resource contention), service dependency anomalies (e.g., version compatibility issues), and data flow exceptions (e.g., message queue accumulation).

There are also other uncertainties such as uncontrollable external inputs (e.g., abnormal user request parameters), inherent randomness introduced by employing Artificial Intelligence (AI) technologies (e.g., prediction fluctuations in machine learning models), and system-internal stochastic function calls (e.g., random load balancing algorithms). These uncertainties exhibit not only diverse types, but also dynamic and concealed characteristics. For instance, transient resource contention at an edge node might only be triggered under specific loads, while version upgrades of dependent services might implicitly introduce protocol parsing errors.

Furthermore, uncertainties exhibit deep coupling across hierarchical levels (ranging from hardware to systems and applications) and cross-service scenarios such as interactions and dependencies between upstream and downstream services in call chains. For example, malicious user inputs might trigger AI model misjudgments, while random load balancing strategies could amplify network jitter effects, potentially triggering cascading failures. Existing approaches that rely on static topology modeling or limited-dimensional monitoring (e.g., analyzing only logs and network metrics) often lack capabilities of dynamically understanding spatial-temporal correlations of multi-dimensional uncertainties. Additionally, these approaches fail to quantify hidden risks from inherent uncertainties (e.g., nondeterministic programs), which leaves test input generation without sufficient guidance.

Therefore, unified modeling and dynamic detection of multi-dimensional uncertainties form the foundational prerequisite to construct uncertainty-oriented fuzzing frameworks and detect complex failure scenarios.

2.2 Dynamic uncertainty propagation

Inter-dependencies among microservices allow uncertainties localized to one service node (e.g., a single service timeout) to rapidly propagate through call chains, message queues, or shared storage, which consequently trigger cascading failures or even system-wide avalanches. For instance, a response delay in an order service might generate a high volume of retry traffic to the payment service, subsequently exhausting database connection pools and potentially paralyzing the entire transaction workflow. Such uncertainty propagation exhibits nonlinear, asynchronous, and path-indeterminate characteristics. Specifically, propagation paths may dynamically change due to load-balancing strategies, states of circuit-breakers (for isolating faulty services), or message queue backlogs. Therefore, it is very difficult to rely on typical metrics (e.g., error rates) to quantify the intensity and boundaries of uncertainty propagation. More critically, observability limitations in distributed systems (e.g., incomplete trace data) can obscure critical propagation nodes, resulting in scenarios where localized anomalies are visible, but impacts remain opaque.

Existing approaches that rely on static dependency graphs or simple error-rate statistics fail to model the causal logic (e.g., triggering conditions for cascading failures) and its implications (e.g., robustness degradation) of uncertainty propagation. Metrics such as Mean Time To Recovery (MTTR) further fall short in quantifying dynamic threats to system resilience (e.g., hidden bottleneck services).

Therefore, we call for research on studying causal inference and assessment of uncertainty propagation of microservices, tracking and quantifying uncertainty propagation and its impact on system quality properties (e.g., availability). Doing so is essential for prioritizing high-risk paths and underpin uncertainty-aware fuzzing methodologies.

2.3 Combinatorial explosion of microservice uncertainties

Industrial microservices systems often require hundreds to thousands of functional services to support complex business logic. For example, the shopping platform developed by Meituan (a Fortune 500 corporation) comprises over 2,000 services [87], and WeChat (the flagship mobile super-app of Tencent, a Fortune 500 corporation) includes more than 3,000 services as of 2018 [93]. These services form a high-dimensional dependency topology: a single business request processing typically triggers multi-level, cross-service and distributed call chains (e.g., rendering a single web page of Amazon, a Fortune 500 corporation, invokes approximately 100–150 APIs [94]). Consequently, the combinatorial space of uncertainties grows exponentially with the number of services. For instance, 3,000 services could yield 2^{3000} abnormal state combinations. Existing testing methods are inefficient at generating high-quality test cases, facing a needle-in-a-haystack dilemma: excessive resources are wasted on low-risk combinations (e.g., isolated anomalies in non-critical services), while high-risk scenarios (e.g., concurrent failures in multiple strongly dependent services within payment workflows) are uncovered.

Existing testing techniques are unable to intelligently balance propagation risks of cross-service combinations of uncertainties, business priorities, and critical scenarios, leading to poor testing efficiency and fault detection rates. Therefore, developing adaptive testing strategies and coverage optimization mechanisms that account for combinations of microservices uncertainties is critical to reconcile the tradeoff between constrained testing resources and combinatorial explosion. This is the pragmatic objective of uncertainty-aware fuzzing research for industrial microservices systems.

2.4 Uncertainty in Microservices and Traditional Distributed Systems

Many uncertainties we discussed above are not unique to microservices but are shared across general distributed systems. Examples such as network latency variability, transient dependency failures, and data inconsistency due to asynchronous

communication are well-known challenges in traditional distributed computing paradigms, including Service-Oriented Architecture (SOA) and client-server systems [42, 73]. Network latency and partial failures are fundamental concerns in distributed systems that drive adoption of failure detectors and consistency protocols. Transient dependency failures and cascading effects have also been studied extensively in the fault tolerance literature, where techniques such as timeouts, retries, and idempotency are used to mitigate their impact [23, 81].

However, due to architectural characteristics, microservices further amplify these uncertainties. For instance, traditional distributed systems [27]. Moreover, each service can scale and evolve independently, which leads to a larger state space of possible configurations and interactions where uncertainties can propagate in non-trivial ways [44].

Recent industrial systems further integrate AI- and LLM-based components into microservice architectures [1], which introduces new forms of uncertainties by interacting with already existing challenges intrinsic to distributed systems. For example, inference latency, model nondeterminism, probabilistic outputs, and model version drift can introduce additional variability in service responses. Specifically, LLM-based services are sensitive to input phrasing, prompt changes, and contextual dependencies. All these may lead to uncertain system behaviors that are difficult to validate. Furthermore, after being embedded in microservice workflows, their stochastic and evolving nature can amplify uncertainty propagation across services, and hence complicate fault localization, and eventually affect the end-to-end reliability of the systems.

Furthermore, AI- and LLM-driven decision-making often relies on external model-serving platforms and large-scale inference infrastructures, which introduces additional dependencies and potential failure modes. For instance, resource contention in GPU clusters, dynamic model updates, and opaque performance characteristics may increase latency fluctuations and cascade failures in microservice systems. Consequently, uncertainty in modern microservices increasingly arises not only from traditional distributed-system factors but also from the inherent variability of learning-based components.

Although many uncertainties are shared with distributed systems in general, along with the advance of AI- and LLM-based techniques, modern microservices introduce higher degrees of variability, emergent cross-service behaviors, and operational complexity that make uncertainty quantification and testing particularly challenging.

3 The State of Art

Microservices testing and uncertainty-aware software engineering in complex software systems (e.g., Cyber-Physical Systems (CPS)) has gained significant attention from both academia and industry in the last decade. In this section, we discuss the state-of-the-art from these aspects: microservices fuzz testing, uncertainty classification and characterization, uncertainty quantification, and uncertainty-aware testing.

3.1 Microservices testing

3.1.1 API fuzzing. Most existing efforts in microservice testing focus on single-service testing, particularly API fuzzing. Currently, mainstream API paradigms for microservices include REST, RPC, and GraphQL. In the rest of the section, we discuss the literature of testing each type of APIs.

REST defines API design guidelines for resource access and manipulation using standard HTTP methods, making it suitable for open platforms and third-party integrations. Its broad adoption has spawned a vast ecosystem of open-source tools, and REST API fuzzing has consequently become the most extensively studied area. Golmohammadi et al. [32] surveyed 92 articles on REST API testing published from 2009 to 2022, and revealed that 72% of the proposed methods belong to black-box testing. Most **REST API black-box testing** approaches leverage API specifications (e.g.,

OpenAPI Specification, OAS, formerly known as Swagger) to parse accessible requests and validate interface behaviors using runtime request/response data. For instance, Restler [10–12, 30, 31] infers API request dependencies from OAS and runtime data to construct test sequences, effectively probing REST API behaviors. Morest [49] formalizes REST API dependencies as property graphs to capture relationships between operations and data structures, enabling test case generation and fault detection. RestCT [80] is a combinatorial testing method that analyzes request/parameter dependencies and constraints via OAS and runtime data, generating test cases to exhaustively cover potential API input combinations. Kim et al. [39, 40] introduced reinforcement learning and NLP-driven techniques to infer API logic and constraints for test generation. Schemathesis [36] is a property-based testing tool that derives test oracles from OAS semantic rules to validate REST API compliance. Rueda et al. [65] and Chen et al. [21] leveraged REST design principles (e.g., idempotency of HTTP methods like GET) to define metamorphic relations for effective test validation. DeepREST [22] overcomes OAS limitations by using curiosity-driven reinforcement learning to uncover hidden API logic and constraints and hence guide test generation. Unlike OAS-based black-box testing methods, **REST API white-box testing** approaches delve into internal system implementations, and combine source code analysis, dynamic execution traces, and metrics such as code coverage, taint analysis, and SQL query interception. For instance, Pythia [9] is an approach that extends Restler with code coverage guidance but requires manual pre-configuration for instrumentation. EvoMASTER [4, 6, 8, 85, 86, 89] fully supports automated white-box fuzzing for APIs. LT-MOSA [69] employs hierarchical clustering of API call sequences to optimize test case generation.

GraphQL was designed to address limitations of REST APIs in data access. For example, GraphQL allows clients to precisely specify what data to retrieve from a graph-like structure [35, 70], which resolves inherent issues in REST such as over-fetching and under-fetching. Similar to REST, GraphQL provides API specifications (e.g., GraphQL Schema) to parse accessible requests, enabling black-box testing without source code access. A recent literature review conducted by [62] highlights the scarcity of GraphQL automated testing research, as the authors managed to identify only five black-box methods [36, 38, 74, 83] and one white-box approach [13, 14] built on top of EvoMASTER. For instance, Hatfield-Dodds and Dygalo [36] proposed to derive structural and semantic rules from GraphQL schema using the Hypothesis framework [50] to generate test cases and oracles. Karlsson and Causevic [38] proposed to extract data types and relationships from schema, employ randomized strategies to generate data or user-defined data to send requests, and then validate responses via HTTP status codes and schema compliance. Zetterlun et al. [83] proposed AutoGraphQL, a tool leveraging real-world GraphQL queries from operations to guide test case generation and detect regressions. Vargas et al. [74] proposed four divergence rules to mutate existing test cases, and then compare the original and variant outcomes to assess API robustness.

Remote Procedure Call (RPC) is a communication protocol and mechanism that enables cross-service, cross-process function or procedure invocations [15]. By abstracting the complexity of underlying communication, RPC makes remote service calls as straightforward as local function calls. Furthermore, RPC frameworks typically employ high-efficiency serialization and deserialization and lightweight communication protocols to minimize network payloads and optimize transmission performance. These features have made RPC a cornerstone for enterprise-scale microservice systems [59]. Unlike REST and GraphQL, RPC lacks a unified API specification. Instead, RPC opts for flexible interface definitions. For instance, RPC frameworks, such as Thrift,¹ gRPC² and Tars,³ have their own interface definition languages (IDL) to define service interfaces, and support the generation of code (e.g., client stub) in multiple programming languages. However,

¹<https://thrift.apache.org/docs/idl>

²<https://protobuf.dev/overview/>

³https://tarscloud.github.io/TarsDocs_en/base/tars-protocol.html

IDL-based API specifications are not a mandatory requirement for RPC implementation. For example, frameworks such as Dubbo,⁴ SOFARPC,⁵ and Mota⁶ support defining and exposing interfaces directly through Java interfaces, configured via framework-specific settings. RPC API testing approaches are currently limited. Notable efforts include the RPC API testing method based on EvoMASTER [87, 89], and Zero-Config proposed by Wang et al. [77]. Zero-Config analyzes Protobuf-based API specifications and uses LibFuzzer to automatically generate test cases for gRPC APIs developed in C++. However, the tool is currently not publicly accessible.

3.1.2 System-level testing of microservices. System-level testing for microservices is relatively scarce, with works mainly focusing on regression testing based on historical data [29, 46, 47], reliability testing [19], and black-box testing based on business logic or formal expressions [37, 51, 61]. Specifically, Liu et al. [47] proposed to automatically generate mocking points by identifying the underlying system state, internal state, and external state of microservice dependencies, aiming to address the issue of online traffic playback failures caused by microservices' dependency on state. Di et al. [24] proposed MicroFuzz to improve code coverage and anomaly detection by using techniques such as dynamic priority seed generation, parallelized multi-stage pipelines, and cross-service context dependency tracking. Gazzola et al. [29] proposed ExVivoMicroTest to generate regression test cases by monitoring and recording runtime data from microservices. Chen et al. [20] proposed optimization methods for selecting regression test cases. Camilli et al. [19] introduced MIPaRT to enable performance and reliability testing of microservices. MIPaRT generates test cases based on real operation data, calculates performance and reliability metrics by monitoring and collecting runtime data from executing test cases, and provides visualizations of collected data to support the continuous evaluation of microservices systems. Hillah et al. [37] proposed a model-based testing approach to generate test cases from pre-defined models (e.g., test configuration model, service interface model, and service behavior model). Quenum and Akinine [61] proposed to generate test cases for microservices based on formal specification statements. To apply metamorphic testing in microservice applications, Luo et al. [51] designed metamorphic relations based on the business logic of each application. Almutawa et al. [2] explored the use of Large Language Models (LLMs) to assist engineers in performing end-to-end microservice testing.

Summary

The literature primarily focuses on testing individual services (i.e., API fuzzing), while system-level testing methods still suffer from a heavy reliance on manual intervention. In industrial practice, system-level testing, integration testing, and end-to-end testing (which begins at the user interfaces and spans the microservices architecture and backend components such as databases to validate complete business processes) also face challenges due to insufficient automation.

3.2 Uncertainty characterization, quantification, and uncertainty-aware testing and optimization

In this section, we discuss studies of uncertainties in software engineering from three aspects: uncertainty classification and characterization, uncertainty quantification and uncertainty-aware testing.

3.2.1 Uncertainty classification and characterization. In the early stages of uncertainty modeling research in the field of software engineering, due to its popularity, the Unified Modeling Language (UML) was chosen as the primary modeling

⁴<https://dubbo.apache.org/en/>

⁵<https://www.sofastack.tech/en/>

⁶<https://github.com/weibocom/motan>

language for specifying uncertainties. For example, Ma et al. [55] proposed a fuzzy UML data modeling approach, which is translated into fuzzy description logic for the correctness verification of fuzzy attributes. Riebisch et al. [64] proposed to model uncertainty in UML use case diagrams, sequence diagrams, and state machines. Motameni et al. [58] proposed to transform UML state machines into fuzzy Petri nets. To support test generation, Garousi et al. [28] modeled temporal uncertainty within UML sequence diagrams. However, these early approaches are preliminary and insufficient in effectively handling complex and multi-dimensional uncertainties.

Subsequently, Zhang et al. [90] proposed U-Model, which defines uncertainty and its related concepts from a software engineering perspective and clarifies their relationships at the conceptual level. Building on this, a more comprehensive UML-based uncertainty modeling language, named UncerTum [84], was proposed. Later, the international standard Precise Semantics for Uncertainty Modeling (PSUM) 1.0 Beta was adopted at OMG⁷, which unifies the understanding of uncertainty in the software engineering domain and provides a standardized and reliable foundation for the development and application of uncertainty modeling.

3.2.2 Uncertainty quantification. In AI-empowered software systems, uncertainties present diversity, covering inherent system complexity, data limitations (e.g., noise, sparsity), dynamic operating environment, etc. Especially, epistemic uncertainty, rooted in incomplete knowledge, subjective judgment and assumptions and insufficient data, increases along with the growth of the system complexity. This type of uncertainties is inherently dynamic and complex, which makes it difficult to be described using fixed probability models. In such cases, Bayesian methods quantify uncertainty by utilizing prior knowledge, updating prior probabilities with new observational data, and generating more accurate posterior probability distributions. These methods have been widely applied in software engineering, such as in model-based testing [17] and performance evaluation of configurable software systems [26]. Monte Carlo simulations use random sampling to generate large numbers of samples to estimate uncertainty distributions. However, when epistemic uncertainty is high, constructing precise probabilistic models becomes challenging, and this method requires substantial computational resources. Information theory methods utilize entropy, mutual information, and relative entropy to quantify the uncertainty and information content of information or data, thus assessing epistemic uncertainty [79]. Fuzzy logic uses fuzzy sets and logical operators to describe the fuzzy attributes and states of things, as well as the fuzzy relationships between them. It plays an important role in modeling and analyzing epistemic uncertainty and has been widely applied in fields such as expert systems, pattern recognition, and decision support systems [25].

Practical applications of AI and machine learning face prediction uncertainties. Meronen systematically studied uncertainty quantification in deep learning models [57], and proposed a Gaussian process-based uncertainty quantification method suitable for resource-constrained environments. To reduce the storage overhead associated with maintaining Markov Chain Monte Carlo (MCMC) samples and demonstrate the practicality of MCMC methods in modern Bayesian neural network applications, Wang et al. [76] used Generative Adversarial Networks (GANs) to simulate MCMC samples and obtain a parameterized approximation of their distribution, thus achieving uncertainty quantification. Srivastava et al. [68] used Monte Carlo Dropout as a regularization term for uncertainty computation to avoid posterior probability calculations. Additionally, methods such as deep Gaussian processes [82] and ensemble-based uncertainty quantification have been applied to quantify prediction uncertainties [48].

In addition, with the rapid development and widespread application of machine learning and AI technologies, several open-source uncertainty quantification tools have been released, such as Uncertainty Toolbox [72], Uncertainty

⁷Precise Semantics for Uncertainty Modeling (PSUM): <https://www.omg.org/spec/PSUM/1.0/Beta1/About-PSUM>

Wizard [78], and TensorFlow Probability.⁸ These tools offer a wealth of algorithms for predicting uncertainty and play an important role in uncertainty quantification, calibration, and visualization.

3.2.3 Uncertainty-aware testing and optimization. Uncertainty-aware testing explicitly acknowledges uncertainties arising from incomplete requirements, complex system interactions, unpredictable user behaviors, probabilistic outputs of AI-driven components, and dynamic operating environment, when testing software systems. At the earlier stage of uncertainty-aware testing, Walkinshaw et al. [75] proposed a black-box testing framework that uses genetic programming to infer the behavioral model of the system under test and selects the most uncertain test cases based on this model. Zhang et al. proposed UcerTest [84], which is a model-based testing approach that automatically generates test cases to test CPS under environmental uncertainties. Zhang et al. also proposed UcerPrio [91] to prioritize high-uncertain test cases for regression testing. For testing self-healing CPS, Ma et al. [54] defined a metric named fragility, based on which an reinforcement learning based testing approach was proposed to test CPS under uncertainties [53].

To automatically generate test oracles for testing the Simulink model of CPS, Menghi et al. [56] used white noise signals to simulate input uncertainties and uncertain parameter values to simulate unknown hardware choices. Haq et al. [34] proposed SAMOTA for generating test sets that consider uncertainty in the predictions of alternative models. Shin et al. [67] proposed a UML-based domain-specific language, HITECS, for Hardware-in-Loop (HiL) test case specification and uncertainty-aware analysis methods to check the well-behavedness of HiL test cases. Camilli et al. [16] proposed a method that dynamically generates test cases using an uncertainty sampling strategy to deal with uncertain parameters in the Markov decision process model of the Tele Assistance System. Camilli et al. [18] also leveraged Markov Decision Processes with beliefs attached to transition probabilities, to enable automated generation and selection of test cases.

Summary

Uncertainty characterization in software engineering have matured with an established metamodel and standard. While various quantification methods exist, there is a notable lack of guidelines for selecting appropriate techniques for specific uncertainty types. Current uncertainty-aware testing approaches predominantly emphasize external uncertainties (e.g., environmental variability) through black-box testing. However, the literature largely neglects systematic methods to address internal uncertainties and their interactions such as component inter-dependencies. Moreover, the literature has limited exploration of modern architectures like microservices, where distributed workflows and service orchestration introduce unique challenges.

4 The Way Forward

After acknowledging the three challenges (Section 2), in this section we present our vision on establishing an uncertainty-driven system-level testing framework for industrial microservices systems, spanning from unified uncertainty modeling, online detection, causal inference, the impact analysis of uncertainty propagation on system quality in microservices, and all the way to continuous uncertainty-driven fuzzing of microservices. The framework is conceptual and has not yet been fully implemented. However, in this paper we aim to identify key components and workflows that are essential for guiding future development and evaluation.

⁸TensorFlow Probability: <https://www.tensorflow.org/probability>

4.1 Architecture

As shown in Figure 2, the overall architecture of our envisioned uncertainty-driven system-level fuzzing framework for industrial microservices can advance the state of art through the following four components:

- *UncerSense* is the component for online detection of microservice uncertainties, based on architectural characteristics of microservices and industrial production data. The framework will be equipped with an uncertainty detection mechanism which dynamically captures uncertainty occurrences during test execution (e.g., real-time detection of uncertainties $u_1^t, \dots, u_i^t, \dots, u_n^t$, where n is the total number of uncertainty occurrences detected by executing test case t).
- *UncerMeter* is the component for quantifying uncertainties and their impact on system quality properties. *UncerMeter* quantifies known uncertainties with fine-grained metrics (e.g., um_i^t denoting the measured uncertainty degree of u_i^t), and assess cascading effects on system quality attributes (e.g., $sm_1^t \dots sm_m^t$ denoting the values of system quality metrics observed when $u_1^t, \dots, u_i^t, \dots, u_n^t$ occur).
- *UncerFuzz* serves as the execution engine for uncertainty-driven microservice testing. It employs a hierarchical, multi- or many-objective optimization algorithm that integrates identified uncertainty occurrences, quantified impacts, and business priorities to generate test cases in favor of high-risk scenarios (e.g., core service call chains) and critical paths.
- *UncerMaster* integrates the above three components and delivers an end-to-end solution for continuous uncertainty detection, uncertainty and impact quantification, and fuzzing.

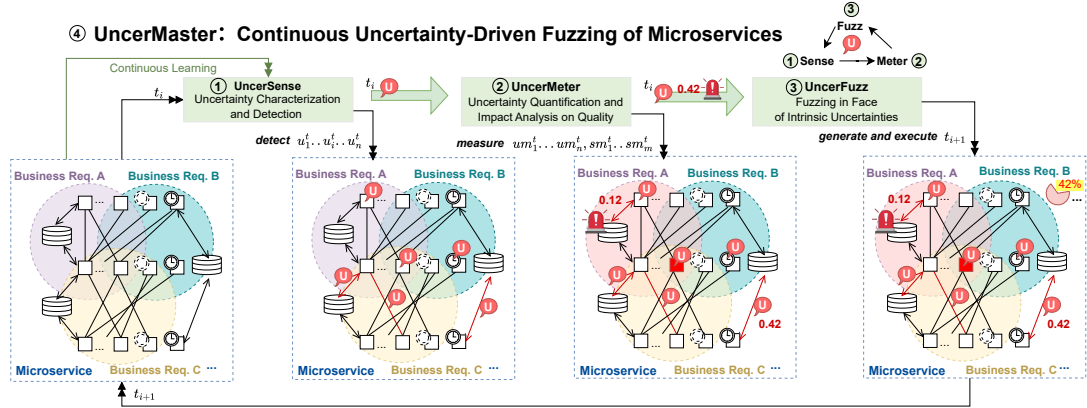


Fig. 2. Overview of Continuous Uncertain-driven Fuzzing of Microservices (UncerMaster)

The rest of the section discusses each component and UncerMaster in detail.

4.2 Uncertainty Characterization and Detection (UncerSense)

4.2.1 Uncertainty classification and characterization. A comprehensive understanding of uncertainties inherent in microservices including their generative mechanisms (e.g., non-deterministic API behaviors), characteristics (e.g., temporal variability), and taxonomies (e.g., input-, propagation-, and environment-induced uncertainties) forms the foundation to develop uncertainty-aware testing methodologies tailored to microservices. To address this gap, we envision a

modeling framework, named Uncertainty-aware Microservices Modeling Language (UncerMML). As recommended practices, UncerMML should be built upon international standards (e.g., PSUM, SysML v2⁹) and stand on the shoulders of existing literature and best practices (e.g., U-Model [90] and EvoMASTER supporting for both black-box and white-box testing [4, 85, 86]), to enable automated detection of multi-dimensional uncertainty sources and their cascading impacts, thereby addressing the critical absence of holistic uncertainty modeling in microservices research.

The first step towards devising UncerMML involves detecting faults from large-scale operation and testing datasets, by systematically applying traffic recording and replay techniques. To achieve this requires industrial collaborations (to get access to large-scale real-world datasets), systematic detecting uncertainties across multiple dimensions by incorporating core characteristics of microservice architectures, such as distributed communication, heterogeneous technology stacks, and dynamic scalability. For instance, at the architectural level, uncertainties can be classified as exogenous (external inputs), interactive (cross-service dependencies), or endogenous (internal system behaviors), and technical implementations may include nondeterministic program behaviors induced by randomized algorithms (e.g., load balancing), message sequence disorders in partitioned queues (e.g., Kafka), and dynamic routing policy drifts in service meshes (e.g., Istio). Microservice uncertainties can be characterized in a fine-grained manner from four interrelated aspects: Uncertainty Type (UT), Source (US), Uncertainty Characteristics (UC_u), and Microservice Characteristics (UC_s). For example, a message ordering anomaly could be categorized as interactive uncertainty (UT), originate from stack misconfigurations (US), exhibit randomness and dynamism (UC_u), and is associated with the complexity of inter-service communication and data consistency requirements in microservice (UC_s). Furthermore, UncerMML should be able to systematically generalize such uncertainty types such as distinguishing epistemic uncertainty (e.g., design flaws in data schemas) from aleatory uncertainty (e.g., inherent randomness like network jitter) to reveal the complete uncertainty propagation from root causes to propagation effects.

4.2.2 Uncertainty detection. Online uncertainty detection in microservices requires the integration of both black-box and white-box techniques to dynamically monitor both behaviors exposed to the external and internal states, thereby addressing complex scenarios arising from diverse uncertainty sources of various natures. Black-box techniques can rapidly detect exogenous uncertainties or explicit interaction issues by monitoring API request-response patterns, interface throughput fluctuations, and anomalous keywords in logs (e.g., timeouts, retries). However, their inability to observe internal states limits the detection of endogenous uncertainties. To overcome this, white-box techniques (e.g., EvoMASTER [8]) can be used to complement black-box methods by instrumenting code to collect fine-grained data (e.g., function call stacks, control flow paths, and variable state changes) to support the detection of endogenous uncertainties.

For different uncertainty types defined in the taxonomy of UncerMML, we need to devise different detection strategies. For uncertainties with explicit sources (e.g., non-deterministic programs), predefined functions related to non-deterministic operations (e.g., `Math.random()` or specific stochastic libraries) can be instrumented with probes to dynamically tag invocations of these functions, enabling real-time detection of code-inherent uncertainties. Additionally, leveraging inherent capabilities of microservice technology stacks (e.g., Istio service mesh for real-time monitoring and distributed tracing), predefined detection rules for scenarios like routing anomalies or traffic spikes can be established. This forms a rule-driven uncertainty detection library to enable systematic uncertainty awareness.

For complex uncertainties with dynamic nature and unclear sources, we need to employ an adaptive learning framework, which can leverage historical data accumulated through traffic recording/replay techniques and already

⁹SysML v2: <https://www.omg.org/spec/SysML/2.0/Beta2/About-SysML>

identified uncertainties to train an initial uncertainty detection model via supervised learning. When previously-unknown uncertainties detected during traffic replay or online monitoring, the system adopts an active learning mechanism to filter high-value samples. These samples are validated and injected into incremental training datasets. Online learning algorithms can then dynamically adjust model parameters, updating uncertainty detection boundaries and feature weights in real time, achieving data-driven uncertainty awareness.

4.3 Uncertainty Quantification and Impact Analysis of Uncertainties on System Quality (UncerMeter)

Regarding uncertainty quantification in microservices, we must address three key challenges: the dynamic propagation of uncertainties, the opacity of causal relationships, and the complexity of assessing system-wide impact of uncertainties on system quality. An effective approach should integrate white-box testing methodologies, established causal modeling theories and techniques, and a range of uncertainty quantification methods. Specifically, it should support the construction of causal inference models and a multi-dimensional framework for comprehensive assessment.

4.3.1 Causal inference for uncertainty propagation. Based on domain knowledge of microservices, we need to systematically analyze dependencies within microservice architectures and exact potential uncertainty propagation paths with code instrumentation, monitoring data, and event logs. Key dependency types in microservices include:

- Explicit invocations: parent-child service dependencies via RPC requests;
- Implicit dependencies: asynchronous coupling through Kafka message generation and consumption.
- Data dependencies: service-level data contention caused by shared database tables.

Specifically, dependencies formed through explicit invocations can be modeled using probabilistic causal graphs, which quantify the conditional dependency strength between services. Dealing with implicit dependencies require the integration of temporal event alignment (e.g., matching time windows for message generation and consumption) to construct temporal causal networks. For asynchronous dependencies such as message queues, transfer entropy (TE) can be employed to quantify the information transfer strength between event sequences of services and identify the direction of causality in asynchronous dependencies (e.g., the one-way influence from producer services to consumer services), such that the magnitude of the TE value directly reflects the strength of uncertainty propagation. For data dependencies, one can leverage data flow provenance techniques to trace propagation paths of shared data states. For instance, for dependencies involving shared resources like databases, a data state transition matrix can be constructed, based on which critical data flow paths can be extracted through techniques such as matrix decomposition, enabling the identification of critical propagation paths. For large-scale industrial services with massive datasets, we need to develop methods for tracing and identifying uncertainty propagation in data dependencies by sampling a small subset of data items.

All in all, dynamic causal graphs can be synthesized to infer dominant uncertainty propagation paths and critical services, such that systematic characterization of uncertainty propagation can be achieved.

4.3.2 Uncertainty quantification. Existing uncertainty quantification methods predominantly rely on mathematical theories and reasoning frameworks such as probability theory, fuzzy set theory, Bayesian networks, entropy theory, and Dempster-Shafer (D-S) evidence theory. However, uncertainty quantification involves multiple factors, including prior data constraints, types, characteristics and sources of uncertainties, which should be specified in UncerMML (Section 4.2.1).

To quantify code-level endogenous uncertainties, we need to understand the sources and behavior of non-deterministic elements in the code, e.g., map random functions to their corresponding probability distribution models. For instance, Java's standard library functions like `Math.random()` follow a uniform distribution, while `Random.nextGaussian()` generates a normal distribution. Third-party libraries (e.g., Apache Commons Math) extend support to complex distributions such as Poisson and exponential distributions. To quantify these uncertainties, code analysis and runtime instrumentation are required to explicitly identify the distribution type and invocation context (e.g., caller methods, parameter conditions). At the service-component level, statistical models can be used to capture conditional dependencies among abnormal events—such as service invocation failures, message disorder, slow SQL queries, and resource contention. To quantify the uncertainty associated with individual services, entropy-based metrics offer a systematic measure of unpredictability and variability in their behavior. For exogenous data uncertainties originating from sensor noise or missing inputs, fuzzy set theory can be applied to quantify membership degrees and confidence intervals, such as defining triangular membership functions.

4.3.3 Impact analysis of uncertainties on system quality. Based on the constructed dynamic causal graphs (recall Section 4.3.1), uncertainties and their propagation paths can be linked to system quality properties such as availability, throughput, and error rate. For example, sensitivity propagation models (e.g., Monte Carlo simulations) can be employed to quantify the cascading amplification effects of localized faults. By simulating scenarios like the propagation of slow database queries through service call chains, and incorporating factors such as service dependency topology and resource contention probabilities, it becomes possible to predict the impact of these faults on throughput degradation or increases in API timeout rates. In addition, historical traffic data can be used to derive quality degradation correlation functions that map operational uncertainties (e.g., message disorder rates) to business-critical quality properties, such as order fulfillment success rates.

We also need to define a multi-dimensional impact assessment index that combines factors such as business criticality weights (e.g., higher weights for payment services, lower for logging services), real-time quality metrics (e.g., latency and error rates), and causal contribution indicators (e.g., the criticality of propagation paths). For instance, index can be calculated by summing up the quality degradation deltas of each microservice weighted by a factor reflecting the centrality of the microservice in the causal graph, i.e., the topological prominence within the service dependency network and its criticality in uncertainty propagation.

4.4 Uncertainty-aware Fuzzing of Microservices (UncerFuzz)

To be uncertainty-aware, during fuzzing, uncertainties must be actively simulated to expose hidden risks and defects, and to evaluate the dependencies and resilience of microservices under such uncertain conditions. In addition, uncertainties must be incorporated into the objectives that guide the fuzzing process. Given the inherent complexity of microservices in the presence of uncertainties, decomposition strategies are essential to break down the testing problem (such as a set of services supporting multiple business requirements) into manageable sub-problems (subsets of services to test), enabling an adaptive fuzzing plan tailored for system-level testing of microservices. Moreover, dynamic fuzzing must be employed to adapt to runtime feedback, evolve testing strategies, and shift priorities of testing objectives, thereby improve the efficiency and effectiveness of fuzzing for each sub-problem. Hence, an uncertainty-aware fuzzing framework for microservices requires to simulate microservice uncertainties intelligently, explicitly integrate uncertainty information into objectives, and adaptively plan and perform fuzzing of microservices, as illustrated in Figure 3.

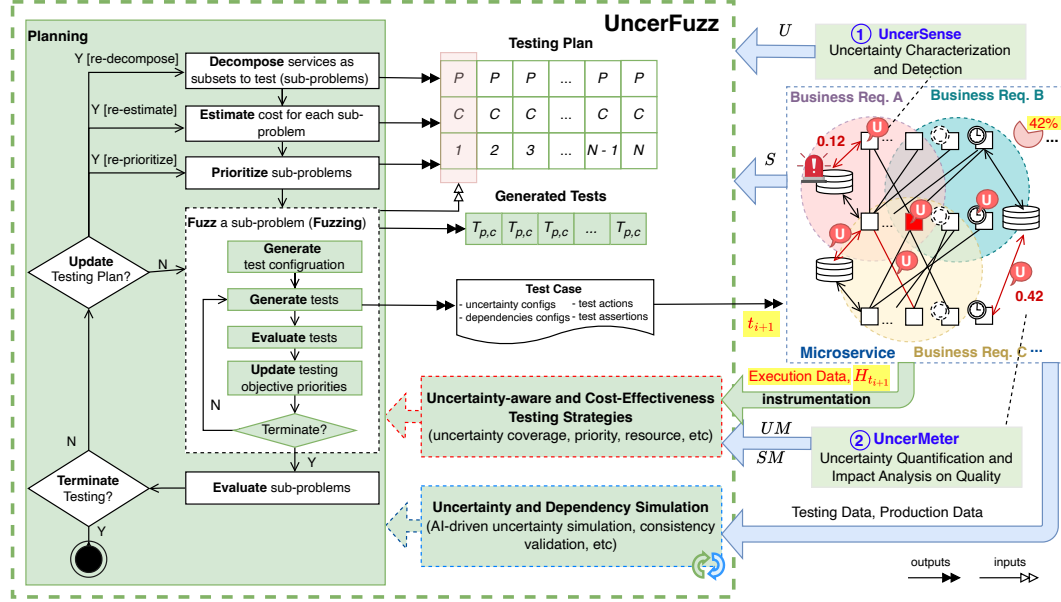


Fig. 3. Overview of Uncertainty-aware Fuzzing of Microservices (UncerFuzz) and its interactions with UncerSense and UncerMeter

4.4.1 Microservice uncertainty simulation. To simulate microservice uncertainties, it is essential to address the practical limitations: the scale of service deployments and the high resource consumption involved (e.g., clearing an interactive database in a test environment taking over five seconds). To overcome these two challenges, service virtualization technologies can be employed to simulate dependent services or databases, enabling efficient testing without fully deploying the entire system and significantly reducing runtime costs.

To enable intelligent system-level testing of microservices, it is necessary not only to identify and simulate service dependencies but also to inject various types of microservice uncertainties during the testing process. Moreover, it is essential to ensure the consistency of simulated multi-dependency behaviors with real-world logic. More specifically, we need to implement the following components:

- **Identification of multi-source dependencies and enabling simulation for the dependencies.** 1) Based on UncerMML and domain knowledge, automatically identify technology stacks related to microservice dependencies, e.g., JDBC connections, message queue communications, and service invocation chains; 2) For each type of dependency, develop simulation techniques and integrate them to the simulation framework. For instance, for JDBC connections, bytecode instrumentation can be used to capture SQL execution statements and templates, which are then used to generate virtual database schemas and query response logic; for asynchronous messaging middleware (e.g., Kafka, RabbitMQ), message publishing can be simulated, partitioning strategies and the injection of consumption delays can be supported as well; 3) Major protocols can be simulated, such as gRPC via service stub generation, GraphQL with virtual response generation, and REST with virtual HTTP response generation from service stubs.

- AI-driven simulation of uncertainties and dependencies.** Heuristics-based and data-driven approaches can be employed to enable intelligent simulation of uncertainties and dependencies in microservices. In API fuzzing, the community has automated the generation of mock objects to handle dependencies of external web services via HTTP [66]. These mock responses are guided by white-box heuristics, such as code coverage. However, in the context of uncertainty-aware microservice fuzzing, both uncertainty and a broad range of dependency types remain to be explored. Besides heuristics-based approaches, (knowledge enhanced) data-driven solutions can also be employed to create or train agents for simulating uncertainties and dependencies in microservices. Using historical traffic recordings (e.g., production data, testing data), multi-modal agents can be trained to handle various simulation tasks, for instance, 1) training sequence-to-sequence models on historical database query results to synthesize mock databases that align with business semantics, 2) employing conditional GANs to generate responses based on request content that conform to business rules for service behavior simulation, and 3) utilizing reinforcement learning to dynamically adjust the mean and variance of response delays to simulate uncertainties, such as network jitter or service overload.
- Consistency validation and resolution.** To ensure the quality of uncertainty and dependency simulation, constraints need to be defined and corresponding solvers need to be implemented for checking various types of consistencies such as data consistency and behavioral contract consistency. For example, in an order creation scenario, if service *A* is simulated to create an order record, the dependent inventory service *B* must update the simulated database to deduct reserved stock, and subsequent queries must return the updated values accordingly.

4.4.2 Hierarchical, uncertainty-aware fuzzing of microservices. Compared to single-service testing, system-level testing of microservices faces significantly more complex challenges. For instance, in a scheduling service, the business logic might span over several functional modules, more than 10 core services (excluding dependent services), and near 1000 critical APIs. The testing process further requires participation from external system actors (e.g., corporate employees) to trigger workflows and provide feedback. Such a system-level testing is large-scale, which must simultaneously address massive combinatorial and multi-dimensional efficiency objectives, rendering existing single-service fuzz testing strategies and algorithms inadequate.

To tackle the dual challenges of combinatorial explosion and resource efficiency in microservice system-level testing, we need a hierarchical, many-objective adaptive testing framework, with the aim to achieve high-efficiency coverage of critical uncertainty scenarios. Details are described below and the key steps and components are also illustrated in Figure 3.

- Uncertainty-aware testing strategies.** Based on real-time detection of microservice uncertainties, establish a mechanism involving multi-dimensional coverage metrics for test generation and optimization with explicit consideration of uncertainties, including, for instance, *uncertainty type coverage* (e.g., time uncertainty covering the dimensions of the past, current, and future), *uncertainty measurement* (e.g., prioritizing fuzzing on endpoints with high entropy, which quantifies the degree of uncertainty in their responses), and *uncertainty coverage* (e.g., proportion of identified uncertainty scenarios exercised). In addition, the strategies incorporate the *priority* and *criticality* of business requirements and associated *risks*, such as dependency path coverage which can be derived from causal graphs to cover uncertainty propagation paths, and weighted critical service coverage with higher weights assigned to core services based on sensitivity analysis results.
- Test problem decomposition.** Given multi-dimensional indicators (e.g., uncertainty metrics, business priorities, risk levels, and critical paths) and the service dependency topology, employ the divide-and-conquer strategy

to partition the testing space into sub-problems, hence transforming a large-scale, complex problem into smaller and more manageable ones. The process can further consider uncertainty correlations and service interaction patterns, which isolate relatively independent uncertainty dimensions and service clusters. Hence, the framework can reduce combinatorial explosion in multi-dimensional uncertainty simulation and avoid to generate redundant or highly overlapping sub-problems. Consequently, such structure-aware decomposition enables more efficient allocation of testing resources and facilitates the incremental and scalable exploration of the system-level testing space.

- **Test cost estimation.** Using dynamic causal graphs and Monte Carlo simulation, estimate the testing cost of each sub-problem. The estimated cost is then used to allocate the fuzzing budget. The cost estimation can incorporate multiple factors, including uncertainty dimensionality, service interaction complexity, historical execution time, and resource consumption. Online feedback from ongoing fuzzing steps can be used to continuously refine cost estimates and correct inaccurate predictions. Such cost estimation can enable informed budget control and help prevent excessive resource consumption during the continued exploration of complex testing scenarios.
- **Test problem prioritization.** Based on the predicted costs, design a scheduling strategy to optimize the order in which sub-problems are solved. Doing so allows for prioritizing simpler and more critical problems, followed by progressively harder or less critical ones. In addition, prioritization can be combined with adaptive sampling and coverage-guided pruning to optimize the exploration of uncertainty configurations. Moreover, by prioritizing computational resources on high-risk and high-impact scenarios, the framework can mitigate high resource demanding required for multi-dimensional uncertainty simulation and combinatorial API interactions, which consequently enables scalable and cost-effective system-level fuzzing in practice.
- **Sub-problem solving (Fuzzing).** Design algorithms to efficiently solve manageable and multi-service fuzzing problems (MSFuzz), along with a many-objective evaluation framework. Such algorithms should integrate intelligent *uncertainty simulation* and *uncertainty-aware testing strategies*. The key steps of the fuzzing is illustrated in Figure 3.
 - With the given a sub-problem – a set of services to test (i.e., P), a budget (i.e., C), identified uncertainties (i.e., U), measured uncertainties (i.e., UM) and system quality indicators (i.e., SM) associated with P , the process begins by generating test configurations to set up the testing environment. This includes tasks such as mocking connected services that are not part of the fuzzing target, initializing required databases, and configuring environmental parameters necessary for simulating uncertainty conditions.
 - Guided by heuristic algorithms (e.g., the Many Independent Objective (MIO) algorithm used in Evo-MASTER [3, 85]) or machine learning techniques (e.g., reinforcement learning), a new test case including configurations of uncertainties and dependencies (see *Test Case* in Figure 3) can be generated during each run.
 - The test is then executed against the services to test (P) in microservices. By integrating white-box techniques, runtime information, such as executed paths, performed interactions with external or internal services, achieved new code coverage, and executed nondeterministic programs, can be collected and used to evaluate the test in terms of testing objectives (see *uncertainty-aware testing strategies* in Figure 3).
 - Based on evaluation results, priorities of testing objectives can be updated dynamically, allowing the fuzzing process to adaptively guide future test generation toward high-impact areas affected by uncertainty.
- **Test planning evaluation.** Evaluate the degree of resolution (e.g., solvability) and trends observed when solving each sub-problem, results of which can then be used to dynamically optimize the overall test plan, such

as by splitting, merging, or pruning sub-problems. The evaluation could also analyze convergence speed, failure discovery rates, and resource utilization patterns, etc., to assess the cost-effectiveness of different testing plans. Specifically, sub-problems that exhibit diminishing benefits or consume excessive computational resources can be further decomposed, consolidated with related sub-problems, or pruned to avoid inefficient use of computational resources. With this feedback-driven evaluation, it can allow adaptive reconfiguration of testing plans and improve cost-efficiency in large-scale uncertainty-driven fuzzing.

4.5 UncerMaster

UncerMaster platform is composed of UncerSense, UncerMeter, and UncerFuzz, providing continuous uncertainty-driven fuzzing of microservices (see Figure 2). Specifically, UncerSense continuously monitors the microservices to infer and detect uncertainties along with their sources. UncerMeter analyzes and measures these uncertainties, tracking their propagation across services and providing metrics that reflect the impact of uncertainty on overall system quality. UncerFuzz enables cost-effective, uncertainty-aware fuzzing of microservices in the presence of uncertainties by integrating with intelligent uncertainty simulation, uncertainty-aware testing objectives, adaptive testing plans, and dynamic fuzzing strategies. Additionally, UncerFuzz can generate new data that feeds back into UncerSense and UncerMeter, enhancing their ability to sense and measure. Together, these components enable UncerMaster to establish a continuous cycle of sensing, measuring, and testing, ensuring adaptive and effective fuzzing that evolves with the microservices.

As a good practice, the envisioned platform should adopt a modular architecture, which decouples core algorithms (e.g., dynamic causal inference engines, many-objective optimizers) from low-level white-box processing (e.g., bytecode instrumentation, traffic recording). The platform should also support flexible extensibility through a plugin-based design. The potential architectural layers are as follows.

- Uncertainty awareness layer: Employ dynamic instrumentation to collect real-time internal states of microservices, and integrate distributed tracing data to construct and dynamically update causal graphs, reflecting real-time uncertainty propagation paths, etc.
- Dependency simulation layer: Simulate dependencies within a microservice technology stack, including Kafka message brokers, database connections, and RPC service stubs.
- Quantitative evaluation layer: Embed an engine for computing values for the uncertainty impact assessment index, which dynamically generates governance priority lists, based on real-time quality metrics and business-critical weights.
- Intelligent testing layer: Integrate multi-phase combinatorial optimization algorithms to enable divide-and-conquer strategies and adaptive fuzzing, and generate simulated datasets via AI techniques and leverage the dependency simulation layer to achieve low-cost, high-fidelity test environments.

5 Illustrating UncerMaster

We use an e-commerce microservice system (see Figure 4 for its overview) as an example to illustrate how UncerMaster operates. Note that the example is intentionally simplified to illustrate how uncertainty could be represented, detected, and acted upon; hence, it is not meant to be functionally complete or semantically correct.

The e-commerce system comprises five services as shown in Figure 4: *Checkout*, *Payment*, *Fraud*, *Order Confirmation*, and *Shipping*. The user-facing entry points are REST APIs in *Checkout* and *Payment*. Internal service-to-service calls

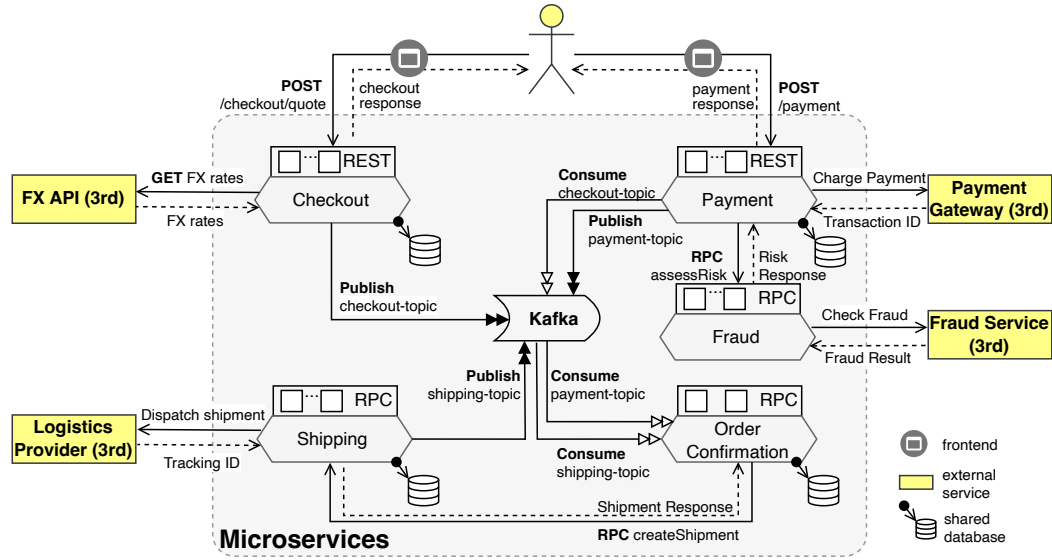


Fig. 4. Microservices and their main interactions in the e-commerce example

are implemented with gRPC (i.e., *Payment* \rightarrow *Fraud* and *Order Confirmation* \rightarrow *Shipping*). Cross-service coordination is event-driven via Kafka topics (i.e., checkout-topic, payment-topic, and shipment-topic). All services persist states in a shared Postgres database. We have open-sourced the running example.¹⁰

A typical flow begins when a *User* requests a checkout quote. The *Checkout* service computes the subtotal, tax, and total, and queries an external foreign-exchange (FX) endpoint (Frankfurter) when a currency conversion is needed. The quote is persisted as a checkout session and then emitted as a checkout-topic event. Next, the *User* submits a payment request to the *Payment* service, which calls the *Fraud* service via gRPC to obtain a risk decision and score. If approved, the *Payment* service invokes a payment gateway client to obtain a transaction identifier. The payment result is persisted and published as a payment-topic event. The *Order Confirmation* service consumes payment-topic events and creates an order-confirmation record. For approved payments, *Order Confirmation* calls the *Shipping* service via gRPC to create a shipment. The *Shipping* service persists the shipment, contacts an external logistics endpoint to dispatch it, and publishes a shipment-topic event. This mixed RPC-event chain decouples services while still forming an end-to-end business flow.

Even in this simplified setting, the system is exposed to heterogeneous sources of uncertainty. Some uncertainties are induced by external dependencies (e.g., FX and logistics endpoints) that may change over time or fail transiently. Some uncertainties are introduced by threshold-based decisions (e.g., fraud scores close to decision boundaries) and by hidden or evolving states (e.g., history-dependent fraud features). Some uncertainties are caused by distributed coordination (e.g., asynchronous consumption and partial progress when an RPC fails after a database write). In the following subsections, we illustrate what UcerMaster would observe and how it would exploit such uncertainty during system-level fuzzing.

¹⁰<https://github.com/man-zhang/microservices-projects>

5.1 Uncertainty Detection with UncerSense

UncerSense plays the role of monitoring runtime information in order to identify uncertainty occurrences during test execution. Such information includes service logs, actuator/prometheus metrics, and interaction metadata across REST, gRPC, Kafka, and database operations. Detected events are then mapped to UncerMML attributes and to predefined modeling assumptions for known uncertainty behaviors. In this paper, we focus on what information requires to be collected to enable identification, rather than on the concrete uncertainty detection mechanisms, which are left as future work and could be rule-based, data-driven, or even hybrid.

In the example, uncertainty can be introduced by input processing, decision logic, external dependencies, and asynchronous propagation. However, uncertainties manifest differently across services and could follow different statistical distributions. For instance, monetary totals in *Checkout* are computed using `BigDecimal` and deterministically rounded to two decimals, which introduces no stochastic numerical variation by itself. Variability is actually primarily driven by time-varying external signals, e.g., periodic refreshes of currency-exchange rates from an external provider (denoted as FX rates),¹¹ and by discrete thresholding such as fraud decisions made by Fraud Service provided by the third party.

For input-processing services, UncerSense collects a lightweight, service-specific record of inputs and computed outputs. In *Checkout*, this includes the request payload (items, quantities, and target currency) and the computed outputs (subtotal, tax, and total). To explain variability, UncerSense also records currency-conversion metadata, such as the exchange-rate value used and whether it was served from cache, refreshed from the live endpoint, or obtained via fallback. Under the same basket, small differences can arise when cached FX rates are refreshed, and larger discontinuities can arise when the service falls back to predefined rates after a failed refresh.

For uncertainty related to external dependencies, UncerSense collects dependency identifiers, timestamps, outcomes, and latency. In the example, main external HTTP dependencies include the *Checkout* call to the Frankfurter FX API, the *Payment* call to a payment gateway, and the *Shipping* call to a logistics provider. Note that, in this example, the payment gateway client in *Payment* is implemented as a local stub that simply generates a transaction ID. However, in real deployments, the external payment gateway is often a major source of uncertainty due to variable latency, transient failures, and provider-side throttling or outages.¹²

For decision-making services, UncerSense captures both the final decision and the intermediate evidence used to reach it. In *Fraud*, the gRPC service computes a score and assigns a discrete decision using fixed thresholds (0.3 and 0.7). The scoring logic also produces an explicit confidence value derived from an ensemble-style predictive distribution. Accordingly, UncerSense records the request features (e.g., `userId` and `amount`), intermediate factors (e.g., history count and external signal), and the resulting score, confidence, and decision. Uncertainty is operationalized as increased decision instability: near decision boundaries, small shifts in features or history may flip the output among APPROVE/REVIEW/REJECT.

For propagated and asynchronous workflows, UncerSense focuses on correlation and state transitions across services. In *Order Confirmation*, UncerSense records correlation keys (e.g., `orderId`, `paymentId`, and `shipmentId`), producer timestamps from events, consumer receive time, and the service-side state transitions written to the database. These records enable identifying partial progress, such as when an order-confirmation record is persisted but the corresponding shipment creation remains unresolved because the gRPC call to *Shipping* fails. In *Shipping*, UncerSense similarly records

¹¹In this paper, "FX" denotes foreign-exchange (currency-exchange) rates. For instance, in the running example, *Checkout* refreshes its cached rates by querying the Frankfurter API (e.g., <https://api.frankfurter.app/latest?from=USD&to=EUR>), as documented at <https://www.frankfurter.app/docs>.

¹²<https://stripe.com/en-nl/resources/more/payment-gateway-down>

dispatch outcomes, response timestamps, and whether a tracking ID is present, since shipments may exist without downstream dispatch completion under failure conditions.

Across all interaction types (service \leftrightarrow service, service \leftrightarrow Kafka, service \leftrightarrow external HTTP, and service \leftrightarrow database), UncerSense can rely on a unified core schema. This schema includes correlation identifiers, timestamps, interaction type, and execution context (service, endpoint/method/topic). Because distributed tracing headers are not propagated by default in the reference implementation, business identifiers serve as the primary linkage for cross-service correlation. Based on these observations, UncerSense dynamically identifies uncertainty instances u_1^t, \dots, u_n^t during the execution of test case t . Each uncertainty instance is associated with its execution context and evidence sources. Potential uncertainties considered in the example are summarized in Table 1 for the illustration purpose and hence it is by no means to be complete. Empirical studies are needed in the future to systematically characterize such uncertainties and develop a comprehensive taxonomy (i.e., UncerMML) for industrial microservice systems.

5.2 Uncertainty Quantification with UncerMeter

After uncertainty instances are detected, UncerMeter quantifies their magnitude and estimates their impact on system-level quality attributes. Once again, all modeling choices below are illustrative and are presented to demonstrate how the framework could operationalize uncertainty in a test loop.

For numerical variation in *Checkout*, UncerMeter can model observed totals as a random variable and estimate distribution parameters from runtime samples. For example, it may fit a Gaussian model and compute μ_x and σ_x from repeated executions under comparable conditions. A non-trivial σ_x then indicates that rate refresh, fallback, and rounding effects can measurably perturb totals. For interaction-induced uncertainty (e.g., external calls and RPCs), UncerMeter can model latency using heavy-tailed distributions such as log-normal. Percentile-based estimators can be used to derive μ_T and σ_T from observed response-time samples. Long-tail latency, such as a 99th percentile exceeding a timeout threshold, can then be related to elevated retries and partial execution. For decision uncertainty in *Fraud*, UncerMeter can treat prediction outputs as stochastic variables around an expected score. The model-provided confidence and dispersion signals can be used to quantify instability near decision boundaries. These measurements can be aggregated per user cohort, amount range, or history regime to support targeted testing. For propagation uncertainty in *Order Confirmation*, UncerMeter can estimate categorical state probabilities of order-confirmation outcomes. A Bayesian model can infer the probability mass of states such as CREATED, FAILED, and SHIPMENT_REQUESTED under different injected conditions.

Regarding uncertainty quantification tools, UncerMeter could integrate existing tools (such as Uncertainty Wizard [78], TensorFlow Probability¹³ and Uncertainty Toolbox [72]) and libraries for uncertainty modeling, measurement, and diagnostics, depending on component type and available telemetry. Table 1 represents the potential applicability of modeling approaches and corresponding tools for quantifying uncertainties. For example, the *Fraud* service employs a neural classifier to output a risk score $y \in [0, 1]$. With Uncertainty Wizard, Monte Carlo dropout is enabled and K stochastic forward passes are executed for the same request, which yields: $\{y^{(k)}\}_{k=1}^K$. The predictive mean $\mu = \frac{1}{K} \sum_k y^{(k)}$ and variance $\sigma^2 = \frac{1}{K} \sum_k (y^{(k)} - \mu)^2$ can be then computed to quantify epistemic uncertainty induced by model stochasticity. We can further compute Bernoulli predictive entropy $H(\mu) = -\mu \log \mu - (1 - \mu) \log(1 - \mu)$ by using the predictive mean μ to capture overall predictive uncertainty and decision ambiguity near classification thresholds. This yields $um_3^t = 1 - \text{confidence}$ in Table 1. For the external logistics dispatch call in *Shipping*, latency L is

¹³TensorFlow: <https://www.tensorflow.org/probability>

modeled as a log-normal random variable. TensorFlow Probability fits parameters (μ_T, σ_T) from runtime samples and computes tail metrics such as $p95(L)$, which are used in um_i^t . This approach is suitable when uncertainty manifests as variability in continuous observations and can be adequately characterized by a known distribution family. For predictive components such as fraud risk models, Uncertainty Toolbox can be used to compute calibration metrics including expected calibration error (ECE), negative log-likelihood (NLL), and Brier score. These diagnostics quantify the reliability of reported confidence values and support threshold selection for uncertainty-driven fuzzing.

Regarding system-level quality metrics, during continuous testing and fuzzing, UncerMeter aggregates execution observations into uncertainty metrics um^t and system-level outcomes sm^t for each test case t . As summarized in Table 1, uncertainty metrics can capture service-local phenomena such as fallback-rate usage and external-call tail latency in *Checkout*, inference uncertainty in *Fraud*, and logistics-dispatch tail latency in *Shipping*. Table 1 also illustrates system-level outcome indicators, including the fraction of payments routed to REVIEW, the fraction of approved payments whose order confirmations remain stuck without shipment association, and the fraction of shipments created without tracking identifiers. Given these measurements, UncerMeter can fit impact models that relate um^t to downstream quality outcomes in sm^t . Table 1 provides an illustrative model where missing tracking identifiers are explained by logistics tail latency, fallback-rate usage, and fraud-model uncertainty, with coefficients fitted from observed executions. These learned relationships are then used by UncerMaster to prioritize uncertainty-driven fuzzing toward the services and interaction paths with the highest inferred impact.

While the above examples illustrate the feasibility of leveraging existing uncertainty quantification approaches or tools, determining which one to use, how to configure it, and how to interpret its outputs in any real-world microservice system remains a non-trivial challenge. The suitability of a given technique depends on factors such as workload characteristics, data availability, latency constraints, and cross-service dependencies. Systematic guidelines for selecting and integrating uncertainty quantification tools in large-scale production systems therefore require further investigation and constitute an important direction for future work.

5.3 Uncertainty-Driven Fuzzing with UncerFuzz

Based on the uncertainty measurements and impact assessments produced by UncerMeter, UncerFuzz aims to generate test cases that prioritize uncertainty-sensitive behaviors and high-impact scenarios. The goal is not to maximize coverage randomly, but to guide exploration toward executions where uncertainty is likely to propagate and expose potential system issues.

In each test cycle, UncerFuzz consumes detected uncertainty instances and their contexts from UncerSense, and quantified uncertainty metrics um_i^t and system-level indicators sm_j^t from UncerMeter. It then constructs uncertainty-aware testing objectives and an adaptive testing plan that reflects both uncertainty severity and cost constraints. Because budgets are limited, UncerFuzz may only actively fuzz a subset of services or call chains at a time, while keeping the remaining parts fixed. Within the selected scope, UncerFuzz synthesizes tests using dynamic fuzzing strategies and intelligent uncertainty simulation. For computation-centric uncertainty in *Checkout*, test generation can emphasize currency selections and execution timing that stress rate caching and fallback behavior. For decision-centric uncertainty in *Fraud*, tests can emphasize inputs that drive inference near decision boundaries, where small perturbations can flip outcomes. For propagation-centric uncertainty in *Order Confirmation* and *Shipping*, tests can emphasize partial-progress executions where persistence succeeds but downstream gRPC calls or external dispatch calls do not complete.

Consider a fuzzing iteration in which UncerFuzz determines the campaign budget to a single service, e.g., *Fraud*, because UncerMeter observes increased instability of inference outcomes near the decision thresholds. In this iteration,

UncerFuzz focuses only on the *Payment* \rightarrow *Fraud* gRPC interaction and treats the payment gateway and downstream order/fulfillment services as out of scope. It constructs a test objective that maximizes decision sensitivity around the thresholds, e.g., maximizing the fraction of executions whose predicted score falls within a narrow band around 0.3 or 0.7 while also maximizing the observed variance of scores and minimizing confidence. To implement this objective, UncerFuzz generates RiskRequest inputs that systematically vary amount and user history conditions, aiming to drive the inferred score into the threshold-adjacent regions where small perturbations can flip APPROVE/REVIEW/REJECT. The test flags a fault indicator when minor input perturbations cause large swings in decision outcomes or when confidence drops sharply without corresponding explanatory changes in input features, which indicates brittle inference behavior that may amplify downstream uncertainty. The resulting traces and quantified metrics are fed back into UncerSense and UncerMeter, enabling UncerFuzz to decide whether to broaden the campaign to downstream services or to refine the inference-focused fuzzing objective.

5.4 Iterative and Continuous Fuzzing with UncerMaster

The UncerMaster platform establishes a continuous cycle of sensing, measuring, and testing by integrating UncerSense, UncerMeter, and UncerFuzz. It orchestrates the end-to-end workflow, coordinates data exchange among the components, and manages iterative testing cycles. During the cycle, as a new uncertainty is observed and quantified, UncerMaster adaptively updates testing objectives, selects where to invest testing budget, and steers fuzzing strategies accordingly. UncerMaster aims to detect uncertainty-induced fault trends before they are triggered at scale by end-user traffic. In our vision, this is achieved by continuously extracting uncertainty characteristics from execution info, quantifying their propagation and impact, and launching focused fuzzing campaigns when risk signals rise. Below, we illustrate two examples to demonstrate how UncerMaster can expose such faults through iterative uncertainty-driven testing.

The first example concerns checkout quoting under an external currency-exchange dependency. Assume that UncerSense observes intermittent timeouts or errors when *Checkout* queries the external currency-exchange endpoint and that quote totals exhibit step changes for identical baskets. UncerSense correlates quote outputs with the applied exchange-rate values and their provenance (cache, live fetch, or fallback), and instantiates uncertainty occurrences u_i^t that reflect external dependency uncertainty interacting with deterministic monetary rounding. UncerMeter then quantifies quote instability and estimates its impact on system-level outcomes such as price consistency. If the estimated risk exceeds a policy threshold, UncerFuzz prioritizes *Checkout* and reproduces the step-change pattern using uncertainty simulation (e.g., injecting controlled failures into the currency-exchange endpoint and varying cache settings).

The second example concerns fulfillment progress along the *Order Confirmation* \rightarrow *Shipping* \rightarrow external logistics chain. Assume that UncerSense observes increasing tail latency and intermittent failures for the external logistics dispatch call in *Shipping*. Also assume that UncerSense observes an increasing rate of approved payments for which *Order Confirmation* persists an order-confirmation record, but the downstream gRPC call to *Shipping* does not complete within an expected time window. To support diagnosis, UncerSense links evidence across services using business identifiers (e.g., orderId and paymentId) and timestamps extracted from REST requests, Kafka events, and gRPC calls, thereby instantiating uncertainty occurrences u_i^t along the propagation path. UncerMeter estimates the distribution of dispatch latency and the probability of partial progress, and derives system-level indicators (e.g., missing shipping progress and missing tracking identifiers). If these indicators exceed policy thresholds, UncerFuzz launches a focused campaign that prioritizes testing towards the *Order Confirmation* and *Shipping* path and steers UncerFuzz to generate tests to cover the path. For example, UncerFuzz can combine injected logistics uncertainty with bursty payment flows

to amplify consumer pressure and timing skew, which increases the likelihood of partial progress and inconsistent cross-service state.

In both examples, faults do not need to be single deterministic bugs in isolated components, but system-level failure modes that emerge when uncertainty interacts with persistence boundaries, asynchronous processing, and downstream calls. The essence of UncerMaster is to treat rising quote instability, partial progress, and inconsistent cross-service states as measurable uncertainty-propagation signals, which are used to proactively construct tests before a large fraction of end-user requests experience the issue. This aligns with our vision that uncertainty-aware system-level testing can serve as an early-warning and pre-development validation loop for industrial microservice systems.

6 Discussions

6.1 Bootstrapping and Cold-Start Challenges

At the beginning, system-level fuzzing may face the “cold-start” problem due to the lack of sufficient observations of system-level failures [92]. With UncerMaster, the framework can initially leverage available domain knowledge, system specifications, architectural dependencies, and coarse-grained monitoring data to construct a preliminary uncertainty and causal model using UncerMML, such as uncertainties associated with input processing, AI/ML-based decision making, and external dependencies. In addition, the framework can enable a bootstrapping phase in which UncerFuzz performs exploratory fuzzing and generates diverse executions under different uncertainty perturbations to provide informative inputs for UncerSense and UncerMeter. Furthermore, seeding is a commonly adopted approach in industrial settings [89], and the initial causal model (e.g., constructed by domain experts) can serve as a seed to guide early-stage fuzzing and analysis.

As testing proceeds, UncerFuzz continuously produces new execution traces, uncertainty injection records, and failure observations. These runtime data are incrementally fed to UncerSense and UncerMeter, which allows them to identify uncertainty and refine uncertainty measurements and causal inferences. Through this iterative feedback loop, the causal model is progressively improved to enable more accurate identification of high-risk behaviors and more effective fuzzing guidance, which eventually may allow the framework to gradually overcome the cold-start limitation.

However, we acknowledge that this iterative refinement process may require substantial time and resources, and its effectiveness may depend on the availability and quality of runtime data. Addressing these practical constraints remains an important future research direction.

6.2 Challenges of Evaluating UncerMaster In Industrial Settings

Setting up and empirically evaluating a system-level testing framework for microservices in industrial settings poses significant challenges. Prior industrial experience with microservice fuzzing (e.g., the EvoMASTER user studies) highlights that even deploying and evaluating a state-of-the-art fuzzer on production-grade services requires non-trivial engineering effort and close collaboration with practitioners, due to issues such as environment setup, dependency management, and integration into existing development workflows [88].

First, industrial microservices may consist of hundreds or thousands of loosely coupled services implemented in diverse technologies and deployed across rapidly changing infrastructures. System-level fuzzing such systems at a realistic scale often demands testing environments provided by the industrial partner. In addition, a major barrier to system-level testing is the prevalence of external and platform dependencies, including payment gateways, authentication services, recommendation engines, inventory services, and various middlewares. Running tests at scale without

disrupting operations commonly requires controlled substitutes (mocks/stubs) [5, 66] as well as systematic *seeding* of system states (e.g., accounts, coupons, inventories, and feature flags) to reach deeper behaviors. An industrial case study [89] on enterprise RPC API fuzzing shows that seeding and mocking are not optional enhancements but practical prerequisites for making white-box fuzzing effective and deployable in enterprise environments.

Beyond test execution, collecting high-quality runtime telemetry such as distributed traces, failure logs, and uncertainty-related measurements, often requires deep integration with production monitoring and observability infrastructures [45]. In industrial environments, such integration may be constrained by security policies, privacy regulations, and operational risks [7]. Moreover, instrumentation itself introduces additional challenges [33, 60]. Although lightweight instrumentation is acceptable in many cases, monitoring overhead and system perturbations may still affect latency-sensitive services, interfere with timing-dependent behaviors, and complicate reproducibility. These effects can lead to spurious findings (e.g., false positives) or hinder reliable replay of test outcomes, thereby requiring careful engineering and further empirical investigation. Thus, the effectiveness of our framework depends on the availability of high-quality traces, logs, and metrics with consistent correlation identifiers across services, as well as principled definitions of system impact (e.g., tail-latency shifts, error-budget consumption, or partial outages). Existing studies on microservices highlight the benefits of such instrumentation and the substantial engineering effort required to obtain actionable and attributable evidence [45]. Finally, fuzzing and its evaluation often necessitates long-running studies to capture rare but high-impact failures, which further increases the required time and resource investment [41, 88].

These realities motivate a staged development and evaluation strategy: (1) begin with a representative subset of services and controlled dependencies (via seeding/mocking), (2) progressively expand the scope and realism of uncertainty and fault scenarios, and (3) collaborate with industry partners to access realistic telemetry and workloads under strict safety controls. Our framework is designed with these constraints in mind, aiming to be incrementally deployable and empirically assessable under practical industrial limitations.

7 Concluding Remarks

Microservices have been widely adopted across various domains, with significant effort dedicated to ensuring their dependability through testing. Most existing testing approaches rely on fuzzing techniques, mostly focusing on API fuzzing, which tests individual services. However, industrial microservices often contain many complex, heterogeneous, dynamic, and interrelated uncertainties that propagate across multiple services, which substantially impact overall system quality. These challenges become increasingly prominent and prevalent as microservices systems inevitably are adopting more and more AI components, which introduces extra layers of uncertainties. These uncertainties have rarely been explicitly acknowledged and systematically integrated into system-level testing strategies for microservices.

In this paper, we argue for the importance and significance of systematically addressing multi-dimensional uncertainties in microservice fuzzing. We propose a concrete vision, named UncerMaster, which is composed of three key components: UncerSense, UncerMeter, and UncerFuzz, by holistically leveraging various types of technologies (e.g., Generative Adversarial Networks, uncertainty quantification), standards (e.g., System Modeling Language, Precise Semantics for Uncertainty Modeling), and best practices (e.g., white-box fuzzing, service virtualization). We also developed an e-commerce example and used it to illustrate UncerMaster systematically. We hope readers find this vision valuable and join us in advancing efforts to realize this framework. Currently, we are developing UncerMaster as an open-source platform, initially targeting JVM-based microservices in collaboration with our industrial partners. However, the platform is designed to be extensible to other programming languages. We are confident that its implementation will

benefit the community and contribute to ensuring the dependability and trustworthiness of industrial microservices, using and building on top of state-of-the-art open-source fuzzers such as EvoMASTER.

Acknowledgments

This work is supported by the National Science Foundation of China (grant agreement No. 62502022). Andrea Arcuri is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

References

- [1] Firas Al-Doghman, Nour Moustafa, Ibrahim Khalil, Nasrin Sohrabi, Zahir Tari, and Albert Y Zomaya. 2022. AI-enabled secure microservices in edge computing: Opportunities and challenges. *IEEE Transactions on Services Computing* 16, 2 (2022), 1485–1504.
- [2] Mustafa Almutawa, Qusai Ghabrah, and Marco Canini. 2024. Towards LLM-Assisted System Testing for Microservices. In *2024 IEEE 44th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 29–34.
- [3] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206.
- [4] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
- [5] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology* 29, 4 (2020), 1–31.
- [6] Andrea Arcuri and Juan P Galeotti. 2020. Testability transformations for existing APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 153–163.
- [7] Andrea Arcuri, Omur Sahin, and Man Zhang. 2025. Fuzzing for Detecting Access Policy Violations in REST APIs.
- [8] Andrea Arcuri, Man Zhang, and Juan Pablo Galeotti. 2024. Advanced White-Box Heuristics for Search-Based Fuzzing of REST APIs. *ACM Transactions on Software Engineering and Methodology* (2024). doi:10.1145/3652157
- [9] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498* (2020).
- [10] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.
- [11] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking security properties of cloud service REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 387–397.
- [12] Efe Barlas, Xin Du, and James C Davis. 2022. Exploiting input sanitization for regex denial of service. In *Proceedings of the 44th International Conference on Software Engineering*. 883–895.
- [13] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based automated testing for GraphQL APIs. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 778–781.
- [14] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2024. Random testing and evolutionary testing for fuzzing GraphQL APIs. *ACM Transactions on the Web* 18, 1 (2024), 1–41.
- [15] Andrew D Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2, 1 (1984), 39–59.
- [16] Matteo Camilli, Carlo Bellettini, Angelo Gargantini, and Patrizia Scandurra. 2018. Online model-based testing under uncertainty. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 36–46.
- [17] Matteo Camilli, Angelo Gargantini, and Patrizia Scandurra. 2020. Model-based hypothesis testing of uncertain software systems. *Software Testing, Verification and Reliability* 30, 2 (2020), e1730.
- [18] Matteo Camilli, Angelo Gargantini, Patrizia Scandurra, and Catia Trubiani. 2021. Uncertainty-aware exploration in model-based testing. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 71–81.
- [19] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. 2022. Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 29–39.
- [20] Li-zhe Chen, Ji Wu, Hai-yan Yang, and Kui Zhang. 2023. A microservice regression testing selection approach based on belief propagation. *Journal of Cloud Computing* 12, 1 (2023), 20.
- [21] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.
- [22] Davide Corradini, Zeno Montolli, Michele Pasqua, and Mariano Ceccato. 2024. DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1383–1394.
- [23] Flavio Cristian. 1991. Understanding fault-tolerant distributed systems. *Commun. ACM* 34, 2 (1991), 56–78.

- [24] Peng Di, Bingchang Liu, and Yiyi Gao. 2024. MicroFuzz: An Efficient Fuzzing Framework for Microservices. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 216–227.
- [25] Alina Diaz-Curbelo, Rafael Alejandro Espin Andrade, and Ángel Manuel Gento Municio. 2020. The role of fuzzy logic to dealing with epistemic uncertainty in supply chain risk assessment: review standpoints. *International Journal of Fuzzy Systems* 22, 8 (2020), 2769–2791.
- [26] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2020. Mastering uncertainty in performance estimations of configurable software systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 684–696.
- [27] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.
- [28] Vahid Garousi. 2008. Traffic-aware stress testing of distributed real-time systems based on UML models in the presence of time uncertainty. In *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 92–101.
- [29] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Marco Mobilio, Itai Segall, Alessandro Tundo, and Luca Ussi. 2023. ExVivoMicroTest: ExVivo testing of microservices. *Journal of Software: Evolution and Process* 35, 4 (2023), e2452.
- [30] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 725–736.
- [31] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 312–323.
- [32] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–41.
- [33] Yasmeen Hammad, Amro Al-Said Ahmad, and Peter Andras. 2025. An empirical study on the performance overhead of code instrumentation in containerised microservices. *Journal of Systems and Software* (2025), 112573.
- [34] Fitash Ul Haq, Donghwan Shin, and Lionel Briand. 2022. Efficient online testing for DNN-enabled systems using surrogate-assisted and many-objective optimization. In *Proceedings of the 44th international conference on software engineering*. 811–822.
- [35] Olaf Hartig and Jorge Pérez. 2018. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference*. 1155–1164.
- [36] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web api schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 345–346.
- [37] Lom Messan Hillah, Ariele-Paolo Maesano, Fabio De Rosa, Fabrice Kordon, Pierre-Henri Wuillemin, Riccardo Fontanelli, Sergio Di Bona, Davide Guerri, and Libero Maesano. 2017. Automation and intelligent scheduling of distributed system functional testing: Model-based functional testing in practice. *International journal on software tools for technology transfer* 19 (2017), 281–308.
- [38] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2021. Automatic property-based testing of graphql apis. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 1–10.
- [39] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing rest api testing with nlp techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1232–1243.
- [40] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2023. Adaptive REST API testing with reinforcement learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 446–458.
- [41] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
- [42] Kathryn B Laskey and Kenneth Laskey. 2009. Service oriented architecture. *Wiley Interdisciplinary Reviews: Computational Statistics* 1, 1 (2009), 101–105.
- [43] Grzegorz Lechowski and Martin Krzywdzinski. 2022. Emerging positions of German firms in the industrial internet of things: A global technological ecosystem perspective. *Global Networks* 22, 4 (2022), 666–683.
- [44] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. *MartinFowler.com* 25, 14-26 (2014), 12.
- [45] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering* 27, 1 (2022), 25.
- [46] Na Li, Jun Wang, Chen Chen, and Hongfei Hu. 2024. Application of API automation testing based on microservice mode in industry software. In *Proceedings of the International Conference on Algorithms, Software Engineering, and Network Security*. 460–464.
- [47] Jiangchao Liu, Jierui Liu, Peng Di, Alex X Liu, and Zexin Zhong. 2022. Record and replay of online traffic for microservices with automatic mocking point identification. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 221–230.
- [48] Jeremiah Liu, John Paisley, Maranthi-Anna Kioumourtoglou, and Brent Coull. 2019. Accurate uncertainty estimation and decomposition in ensemble learning. *Advances in neural information processing systems* 32 (2019).
- [49] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API testing with execution feedback. In *Proceedings of the 44th International Conference on Software Engineering*. 1406–1417.
- [50] Andreas Löschner and Konstantinos Sagonas. 2018. Automating targeted property-based testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 70–80.
- [51] Gang Luo, Xi Zheng, Huai Liu, Rongbin Xu, Dinesh Nagumothu, Ranjith Janapareddi, Er Zhuang, and Xiao Liu. 2020. Verification of microservices using metamorphic testing. In *Algorithms and Architectures for Parallel Processing: 19th International Conference, ICA3PP 2019, Melbourne, VIC, Australia, December 9–11, 2019, Proceedings, Part I* 19. Springer, 138–152.

- [52] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. 2022. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3901–3914.
- [53] Tao Ma, Shaikat Ali, and Tao Yue. 2019. Modeling foundations for executable model-based testing of self-healing cyber-physical systems. *Software & Systems Modeling* 18 (2019), 2843–2873.
- [54] Tao Ma, Shaikat Ali, Tao Yue, and Maged Elaasar. 2019. Testing self-healing cyber-physical systems under uncertainty: a fragility-oriented approach. *Software Quality Journal* 27 (2019), 615–649.
- [55] Zong Min Ma, Fu Zhang, and Li Yan. 2011. Fuzzy information modeling in UML class diagram and relational database models. *Applied Soft Computing* 11, 6 (2011), 4236–4245.
- [56] Claudio Menghi, Shiva Nejati, Khoulood Gaaloul, and Lionel C Briand. 2019. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 27–38.
- [57] Lassi Meronen. 2023. *Uncertainty Quantification in Deep Learning*. Ph.D. Dissertation. Aalto University.
- [58] H Motameni, I Daneshfar, J Bakhshi, and H Nematzadeh. 2010. Transforming fuzzy state diagram to fuzzy Petri net. *Journal of Advances in Computer Research* 1, 1 (2010), 29–44.
- [59] Sam Newman. 2021. *Building microservices: designing fine-grained systems*. "O'Reilly Media, Inc."
- [60] Anders Nöu, Satcheendra Talluri, Alexandru Iosup, and Daniele Bonetta. 2025. Investigating Performance Overhead of Distributed Tracing in Microservices and Serverless Systems. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. 162–166.
- [61] Jose G Quenum and Samir Aknine. 2018. Towards executable specifications for microservices. In *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 41–48.
- [62] Antonio Quiña-Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés. 2023. GraphQL: a systematic mapping study. *Comput. Surveys* 55, 10 (2023), 1–35.
- [63] Mark Richards and Neal Ford. 2025. *Fundamentals of Software Architecture: A Modern Engineering Approach*. "O'Reilly Media, Inc."
- [64] Matthias Riebsch, Ilka Philippow, and Marco Götze. 2002. UML-based statistical test case generation. In *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*. Springer, 394–411.
- [65] Sergio Segura Rueda, José Antonio Parejo Maestre, Javier Troya Castilla, and Antonio Ruiz Cortés. 2017. Metamorphic Testing of RESTful Web APIs. (2017).
- [66] Susruthan Seran, Man Zhang, Onur Duman, and Andrea Arcuri. 2025. Handling Web Service Interactions in Fuzzing with Search-Based Mock-Generation. *ACM Transactions on Software Engineering and Methodology* (2025).
- [67] Seung Yeob Shin, Karim Chaouch, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C Briand, and Frank Zimmer. 2021. Uncertainty-aware specification and analysis for hardware-in-the-loop testing of cyber-physical systems. *Journal of Systems and Software* 171 (2021), 110813.
- [68] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.
- [69] Dimitri Stallenberg, Mitchell Olthoorn, and Annibale Panichella. 2021. Improving test case generation for rest apis through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 117–128.
- [70] Ruben Taelman, Miel Vander Sande, and Ruben Verborgh. 2018. GraphQL-LD: linked data querying with GraphQL. In *ISWC2018, the 17th International Semantic Web Conference*. 1–4.
- [71] Edge Delta Team. 2024. How Many Companies Use Cloud Computing in 2025? [10 Statistics and Insights]. <https://edgedelta.com/company/blog/how-many-companies-use-cloud-computing>.
- [72] Kevin Tran, Willie Neiswanger, Junwoong Yoon, Qingyang Zhang, Eric Xing, and Zachary W Ulissi. 2020. Methods for comparing uncertainty quantifications for material property predictions. *Machine Learning: Science and Technology* 1, 2 (2020), 025006.
- [73] Maarten Van Steen and Andrew S Tanenbaum. 2017. *Distributed systems*. Maarten van Steen Leiden, The Netherlands.
- [74] Daniela Meneses Vargas, Alison Fernandez Blanco, Andreina Cota Vidaurre, Juan Pablo Sandoval Alcocer, Milton Mamani Torres, Alexandre Bergel, and Stéphane Ducasse. 2018. Deviation testing: A test case generation technique for GraphQL APIs. In *11th International Workshop on Smalltalk Technologies (IWST)*. 1–9.
- [75] Neil Walkinshaw and Gordon Fraser. 2017. Uncertainty-driven black-box test data generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 253–263.
- [76] Kuan-Chieh Wang, Paul Vicol, James Lucas, Li Gu, Roger Grosse, and Richard Zemel. 2018. Adversarial distillation of bayesian neural network posteriors. In *International conference on machine learning*. PMLR, 5190–5199.
- [77] Wei Wang, Andrei Benea, and Franjo Ivancic. 2023. Zero-config fuzzing for microservices. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1840–1845.
- [78] Michael Weiss and Paolo Tonella. 2021. Uncertainty-wizard: Fast and user-friendly neural network uncertainty quantification. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 436–441.
- [79] Michael Weiss and Paolo Tonella. 2023. Uncertainty quantification for deep neural networks: An empirical comparison and usage guidelines. *Software Testing, Verification and Reliability* 33, 6 (2023), e1840.
- [80] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial testing of restful apis. In *Proceedings of the 44th International Conference on Software Engineering*. 426–437.

- [81] Liudong Xing. 2020. Cascading failures in Internet of Things: Review and perspectives on reliability and resilience. *IEEE Internet of Things Journal* 8, 1 (2020), 44–64.
- [82] Zhaoyi Xu, Yanjie Guo, and Joseph Homer Saleh. 2021. Accurate remaining useful life prediction with uncertainty quantification: a deep learning and nonstationary gaussian process approach. *IEEE Transactions on Reliability* 71, 1 (2021), 443–456.
- [83] Louise Zetterlund, Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2022. Harvesting production graphql queries to detect schema faults. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 365–376.
- [84] Man Zhang, Shaukat Ali, Tao Yue, Roland Norgren, and Oscar Okariz. 2019. Uncertainty-wise cyber-physical system test modeling. *Software & Systems Modeling* 18 (2019), 1379–1418.
- [85] Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021).
- [86] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–38.
- [87] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-box fuzzing RPC-based APIs with EvoMaster: An industrial case study. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–38.
- [88] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, Kaiming Xue, Zhao Wang, Jian Huo, and Weiwei Huang. 2025. Fuzzing Microservices: A Series of User Studies in Industry on Industrial Systems with EvoMaster. *Science of Computer Programming* (2025), 103322. doi:10.1016/j.scico.2025.103322
- [89] Man Zhang, Andrea Arcuri, Piyun Teng, Kaiming Xue, and Wenhao Wang. 2024. Seeding and mocking in white-box fuzzing enterprise rpc apis: An industrial case study. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2024–2034.
- [90] Man Zhang, Bran Selic, Shaukat Ali, Tao Yue, Oscar Okariz, and Roland Norgren. 2016. Understanding uncertainty in cyber-physical systems: a conceptual model. In *Modelling Foundations and Applications: 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings 12*. Springer, 247–264.
- [91] Man Zhang, Jiahui Wu, Shaukat Ali, and Tao Yue. 2023. Uncertainty-Aware Test Prioritization: Approaches and Empirical Evaluation. *arXiv preprint arXiv:2311.12484* (2023).
- [92] Shenglin Zhang, Sibao Xia, Wenzhao Fan, Binpeng Shi, Xiao Xiong, Zhenyu Zhong, Minghua Ma, Yongqian Sun, and Dan Pei. 2025. Failure diagnosis in microservice systems: A comprehensive survey and analysis. *ACM Transactions on Software Engineering and Methodology* 35, 1 (2025), 1–55.
- [93] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 149–161.
- [94] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Table 1. Potential sources and types of uncertainty in the e-commerce example

Service	System Context	Description
Checkout	Input Processing; External Dependency	<p>Price, tax, and currency conversion are computed with deterministic rounding, while variability is driven by currency-exchange rate refresh, caching, and fallback behavior.</p> <p>UT: Input- and environment-induced uncertainty.</p> <p>US (Aleatory): Time-varying currency-exchange rates and transient failures when calling the Frankfurter API.</p> <p>US (Epistemic): Rate freshness and provenance (cache vs. live vs. fallback) unless explicitly instrumented.</p> <p>UC_u: Temporal variability in quoted totals under identical baskets.</p> <p>UC_s: Currency-exchange client, cache TTL, scheduled refresh with retries, and fallback-rate path.</p> <p>Modeling/Measurement: x as a distribution over observed totals across time and refresh regimes.</p> <p>Example um (hypothetical): $um_1^t = P(\text{rateSource} = \text{fallback}) = 0.12$, $um_2^t = p95(L_{fx}) = 1.8s$.</p> <p>Quantification Tools: Actuator/Prometheus metrics and logs, TensorFlow Probability for distribution fitting (e.g., tail latency).</p>
Payment	Communication; Service Dependency	<p>Payment processing depends on a gRPC call to <i>Fraud</i> to obtain a risk decision, and it emits payment-topic events for downstream processing.</p> <p>UT: Interaction- and decision-induced uncertainty.</p> <p>US (Aleatory): Variable gRPC latency and transient failures in service-to-service communication.</p> <p>US (Epistemic): Limited visibility into downstream decision rationale and evolving history-dependent features.</p> <p>UC_u: Status instability when risk scores are near decision thresholds and when dependency health fluctuates.</p> <p>UC_s: gRPC dependency on <i>Fraud</i> and event publication to Kafka.</p> <p>Modeling/Measurement: T as a distribution over request latency and S as a categorical payment outcome.</p> <p>Example sm (hypothetical): $sm_{\text{review}}^t = P(\text{paymentStatus} = \text{REVIEW})$.</p> <p>Quantification Tools: Actuator/Prometheus metrics and logs, TensorFlow Probability for latency modeling.</p>
Fraud	AI/ML Inference; External Signal	<p>Risk scoring follows an AI/ML inference workflow that outputs a score and an uncertainty indicator (e.g., confidence), which is then mapped to a discrete decision via fixed thresholds.</p> <p>UT: Inference- and decision-induced uncertainty.</p> <p>US (Epistemic): Model approximation, limited/biased training data, and concept drift that reduce fidelity to real fraud dynamics.</p> <p>US (Aleatory): Noisy and time-varying behavioral signals and external assessments in realistic deployments.</p> <p>UC_u: Predictive dispersion and decision flips near thresholds (0.3 and 0.7).</p> <p>UC_s: Model inference pipeline, feature dependency on historical state, and threshold-based post-processing.</p> <p>Modeling/Measurement: y as a predictive distribution, with uncertainty summarized by variance/entropy and confidence intervals.</p> <p>Example um (hypothetical): $um_3^t = 1 - \text{confidence} = 0.27$.</p> <p>Quantification Tools: Uncertainty Wizard for predictive distributions (e.g., MC dropout), and Uncertainty Toolbox for calibration metrics; Actuator/Prometheus metrics and stored fraud-check records support telemetry.</p>
Order Confirmation	Event-Driven Aggregation; Communication	<p>The service consumes payment-topic events, persists an order-confirmation record, and calls <i>Shipping</i> via gRPC for approved payments.</p> <p>UT: Propagation- and partial-progress uncertainty.</p> <p>US (Epistemic): Partial observability of event timing, consumer backlog, and failure points across a multi-step workflow.</p> <p>US (Aleatory): Broker delay, consumer lag, and transient RPC failures to <i>Shipping</i>.</p> <p>UC_u: Ambiguous intermediate states when a DB write succeeds but the downstream gRPC call fails to complete.</p> <p>UC_s: At-least-once event delivery, idempotence checks, and eventual consistency across DB and RPC effects.</p> <p>Modeling/Measurement: S as a categorical distribution over confirmation states and D as a distribution over end-to-end delays.</p> <p>Example sm (hypothetical): $sm_{\text{stuck}}^t = P(\text{approvedPayment} \wedge \text{shipmentId} = \emptyset)$.</p> <p>Quantification Tools: Actuator/Prometheus metrics, application logs, and database state snapshots.</p>
Shipping	Communication; External Dependency	<p>Shipment creation is triggered via gRPC, persisted immediately, and followed by an external logistics dispatch call that may be slow or fail.</p> <p>UT: Environment- and propagation-induced uncertainty.</p> <p>US (Aleatory): Stochastic logistics endpoint latency and availability.</p> <p>US (Epistemic): Limited visibility into external carrier state and dispatch completion.</p> <p>UC_u: Missing-trackingId outcomes and delayed dispatch completion under failure.</p> <p>UC_s: Persist-then-dispatch workflow and external HTTP integration.</p> <p>Modeling/Measurement: T as a long-tail distribution over dispatch latency and M as a missingness indicator for tracking IDs.</p> <p>Example um/sm (hypothetical): $um_4^t = p95(L_{\text{logistics}}) = 3.5s$, $sm_{\text{missingTrack}}^t = P(\text{trackingId} = \emptyset)$.</p> <p>Quantification Tools: Actuator/Prometheus metrics, logs, database state inspection, and TensorFlow Probability for latency modeling.</p>
Cross-Service	Impact Analysis	<p>Impact model (hypothetical): $sm_{\text{missingTrack}}^t \approx \beta_1 p95(L_{\text{logistics}}) + \beta_2 P(\text{rateSource} = \text{fallback}) + \beta_3 (1 - \text{confidence})$, where $\beta_1, \beta_2, \beta_3$ are fitted from observations and used to prioritize uncertainty-based fuzzing.</p>