Idempotency in Service Mesh: For Resiliency of Fog-Native Applications in Multi-Domain Edge-to-Cloud Ecosystems

Matthew Whitaker[1][a], Bruno Volckaert[2][b] and Mays Al-Naday[1][c]
[1]*School of Computer Science and Electrical Engineering University of Essex, U.K.*
[2]*Ghent University - imec, IDLab, Department of Information Technology, Gent, Belgium*

Keywords: Microservices Architecture, Multi-Domain Cloud Systems, Resiliency, Cloud-Native Applications, Service Mesh.

Abstract: Resilient operation of cloud-native applications is a critical requirement to service continuity, and to fostering trust in the cloud paradigm. So far, service meshes have been offering resiliency to a subset of failures. But, they fall short in achieving idempotency for HTTP POST requests. In fact, their current resiliency measures may escalate the impact of a POST request failure. Besides, the current tight control over failures - within central clouds - is being threatened by the growing distribution of applications across heterogeneous clouds. Namely, in moving towards a fog-native paradigm of applications. This renders achieving both idempotency and request satisfaction for POST microservices a non-trivial challenge. To address this challenge, we propose a novel, two-pattern, resiliency solution: *Idempotency* and *Completer*. The first is an idempotency management system that enables safe retries, following transient network/infrastructure failure. While the second is a FaaS-based completer system that enables automated resolution of microservice functional failures. This is realised through systematic integration and application of developer-defined error solvers. The proposed solution has been implemented as a fog-native service, and integrated over example service mesh Consul. The solution is evaluated experimentally, and results show considerable improvement in user satisfaction, including 100% request completion rate. The results further illustrate the scalability of the solution and benefit in closing the current gap in service mesh systems.

1 INTRODUCTION

The cloud-native paradigm is being adopted in developing digital services and networks, including 6G (Netflix, 2018). So far, cloud-native applications (CNAs) have been operating within a single cloud data centre, and connected by the internal network. Moving towards edge-to-cloud computing (a.k.a. fog computing) fosters higher distribution of a CNA's over multiple autonomous cloud systems (ACSes), which connect over an external network. This means the resiliency of distributed CNAs will require cooperation and coordination across multiple ACSes. Cooperation through multi-cluster service peering is emerging with the Multi-Cluster Service (MCS) API, but it is limited to peering provisions at infrastructure layer with no knowledge or management of service failures (Kubernetes, 2024).

Service meshes enable microservice peering across clusters of the same cloud, as well as across ACSes (Gattobigio et al., 2022). In particular, the Consul service mesh enables secure microservice peering through mutual TLS across multiple clouds (Consul, 2024). Moreover, meshes enable a range of resiliency measures to a subset of microservice failures, including: 'circuit breakers', 'retries' and 'bulkheads'. However, existing service meshes lack the ability to achieve idempotency of a request to a HTTP POST microservice. This is because they do not have the ability to differentiate the cause of the request failure and whether this is due to infrastructure disruption, and hence it is safe to retry to a fallback instance, or the failure is caused by a microservice defect that must be resolved first, before retrying. More so, implementing some of the measures above when the failure is due to a microservice defect risks creating what we define as an *idempotency error storm*.

An idempotency error occurs when a non-idempotent HTTP POST request is interrupted by an error, resulting in a request retry that has a different side effect from the original request. Attempting to retry without a prior state can lead to undefined be-

[a] https://orcid.org/0009-0009-1146-1519
[b] https://orcid.org/0000-0003-0575-5894
[c] https://orcid.org/0000-0002-2439-5620

Whitaker, M., Volckaert, B. and Al-Naday, M.
Idempotency in Service Mesh: For Resiliency of Fog-Native Applications in Multi-Domain Edge-to-Cloud Ecosystems.
DOI: 10.5220/0013293900003950

In *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025)*, pages 182-189
ISBN: 978-989-758-747-4; ISSN: 2184-5042

haviour, leaving the system in an unknown state or creating duplicate resources. An *idempotency error storm* is a compounding variation of random unknown states that has an undetermined likelihood of booking duplicate resources (i.e. nondeterministic behaviour), as a result of the compounding number of retries, which may occur in a microservices-based application at different levels of the call stack (Microsoft, 2022). The prevention of idempotency storms is critical in saving resources from duplicate reservations.

Currently, preventing idempotency storms and guaranteeing 'exactly-once' semantics requires disabling the 'retries' measures in service meshes. However, this eliminates gains from retries and automatic request satisfaction when the failure is due to infrastructure disruption, not a microservice defect.

This work proposes a novel, two-pattern, resiliency solution that provides idempotency with automatic request completion for POST microservices. Idempotency is achieved through differentiation of failures caused by infrastructure disruption, from those caused by a microservice defect. The former are allowed retries by the service mesh as they are deemed 'safe', while the latter are prevented from retrying before the microservice defect is resolved. The Completer facilitates automatic resolution of microservice defects and request completion. This is achieved through a FaaS-based application of developer-defined error solvers. Notably, The completer does not interfere or tamper with the application function, but merely facilitate a pathway to implement the developer resolvers. The range of microservice defects that the completer can handle, solely depends on the comprehension of solvers offered by the application provider.

Both the Idempotency Manager (IM) and Completer system are programming language-agnostic, allowing them to interact with a variety of fog-native applications on top, and multiple service meshes underneath. The proposed solution has been prototyped as Java microservices and GoLang functions, integrating with Consul mesh and operating over Kubernetes cluster(s), although the design is not tied to an specific programming language or Service Mesh implementation or container orchestration tool. The solution has been evaluated experimentally and results illustrate 100% user satisfaction, while averting considerable resource waste with idempotency-safe retries.

The remainder of this paper is structured as follows: Section 2 reviews state of the art related work and current implementations to achieve idempotency with further information given in GitHub [1]. Section 3 presents our proposed two-part solution, describing the IM and Completer systems. Section 4 evaluates the proposed solution with a realistic sample fog-native application, distributed over multiple clusters. Finally, Section 5 draws our conclusions and the foreseen future work.

[1]https://github.com/M-Whitaker/closer-idempotency-in-service-mesh-apps

2 RELATED WORK

Resiliency of edge-cloud ecosystems have been investigated comprehensively, covering infrastructure, applications and systems architectures (Amiri et al., 2023; Shahid et al., 2021; Prokhorenko and Ali Babar, 2020). Application resilience generally focuses on recovering task executions to successfully complete a request (Amiri et al., 2023; Mendonca et al., 2020). In a cloud-native environment, the latter breaks down to recovery of inter-microservice communication and operation, disrupted by a transient infrastructure failure, and, of microservice operation caused by functional failure of the microservice itself.

Service meshes have been developed largely to provide resilience to the application through patterns such as *fail-safe* and *retries* (Mendonca et al., 2020; Services, 2024). The works of (Saleh Sedghpour et al., 2022; Karn et al., 2022) provide a systematic benchmark analysis of the circuit breaking and retry patterns in Istio service mesh, formulating a set of configurations for these patterns found to be most efficient. However, their works does not cover challenges of POST idempotency, or how these patterns behave in case of microservice functional failures. The work of (Chandramouli, 2022; de O. Júnior et al., 2022) provides guidelines for leveraging service mesh resilience patterns in DevSecOps but, similar to other work, it does not tackle idempotency issues. The work of (Furusawa et al., 2022) proposes a service mesh controller to actively counter transient failure of infrastructure due to load imbalance. While their work provides a solution for redirecting requests to a different cluster, it does not address functional failure challenges.

Less efforts have tackled functional (non-transient) failures. Example implementations have proposed to re-process failed requests, identified by their idempotency keys after a functional error is fixed (Leach, 2017). However often, complicated (non-reusable) logic is needed to achieve eventual consistency across multiple services because of the types of errors often found in distributed systems (Gabrielson, 2019). Added to that, having error-solver logic concentrated in one codebase violates principles of service decoupling and modularity.

3 PROPOSED SYSTEMS

3.1 Overall Design and Example Application

Here, we define a resiliency solution for FNAs comprised of: an *Idempotency system* and a *Completer*. The first introduces idempotency within a service mesh, enabling safe automatic retries of POST requests that failed due to infrastructure/network disruption while preventing retries of failed requests due to microservice defects. The Completer provides a systematic approach to resolving microservice errors and completing failed POST requests asynchronously, using data collected from the IM. Both systems are built on top of the Consul [2] service mesh, as an example, with ports to other Envoy-based service meshes possible. The design only relies on the fact that each microservice has its own unique Envoy proxy, intercepting all incoming requests.

Figure 1 depicts the proposed solution within an example ecosystem, provides a sample FNA.

3.1.1 Fog Native Application (FNA) Integration

The sample FNA consists of four microservices: a central API gateway, *booking*, *payments*, and *restaurant*. The application is spread over multiple clusters connected via Consul cluster peering. The payment microservice has additional instances, possibly deployed in multiple clusters. The restaurant microservice is a CRUD API for managing the configuration for different restaurants in the system. This configuration can be used to determine business logic in other microservices. The *booking* microservice depends on both its *restaurant* and *payment* counterparts. It is important that the payment microservice is *idempotent*, so that end-users do not incur duplicate payments and the FNA does not include duplicate payment state. It is also a requirement that the payment microservice responds positively to the booking counterpart to *complete* the booking transaction and satisfy the end-user request.

3.1.2 The Idempotency System

The IM is formed of three components: a **LUA filter** associated with the Envoy of each microservice; the **idempotency manager (IM)**; and, the **idempotency information base**. The LUA filter intercepts POST requests and responses on the upstream Envoy. The filter extracts data from the requests, calls out to the IM and interprets the response back, deciding on the logic to be run next. The first part involves extracting the request method and necessary headers, such as `x-request-id` and `completer-downstream-resource`.

This information is used to determine whether the filter is required to run on the request or not. After this, a request digest can be calculated by creating a hash of the request body and base64 encoding it. The request digest is then sent to the IM to check if an already existing request has the same key. The message to the IM additionally includes: the idempotency key (`x-request-id`), the service name and the cluster name. These are needed to ensure that users do not send multiple requests with the same idempotency key and to enable idempotency for multiple services in the same call chain. The cluster name allows each cluster to manage its own idempotency state.

The second part of the filter interprets the response from the idempotency manager, confirming if the request is new or has been processed before. If the request has not been processed before, the IM returns a status code of `404`. The request details are then saved by Envoy to be used on exit of the application, to store the idempotency key for the request. As seen in Figure 2, the request is then forwarded as normal to the application. If the request has been processed before, the idempotency service returns a status code of `200` with all the details of the first response, sent originally by the application. The filter then constructs a response from the information returned by the IM and returns this to the client, as if the response was from the application itself, as seen in Figure 3. The response headers also include two additional fields: `idempotency-status` with a value of `true`, and `x-envoy-ratelimited` with a value of `true`. These response headers are used to inform the client that the response sent back is from the idempotency manager and not the application itself. The client can then choose either to react accordingly, or leave it to the system to resolve the issue. If an error occurs in the filter, a `500` response is returned to the client with the `idempotency-status` header set to `false`, informing the client of a problem with the filter.

The final part of the filter is run after the application has completed. It calls out to the IM to store the idempotency data for the request, which then says that this request has been processed. On a `201` response from the IM the filter returns to the client as normal. Notably, for any response from the IM that indicates an error, the filter logs the error so that it can be processed by the observability system.

The idempotency manager (IM) is a RESTful microservice with two endpoints that retrieve and save

[2] https://www.consul.io

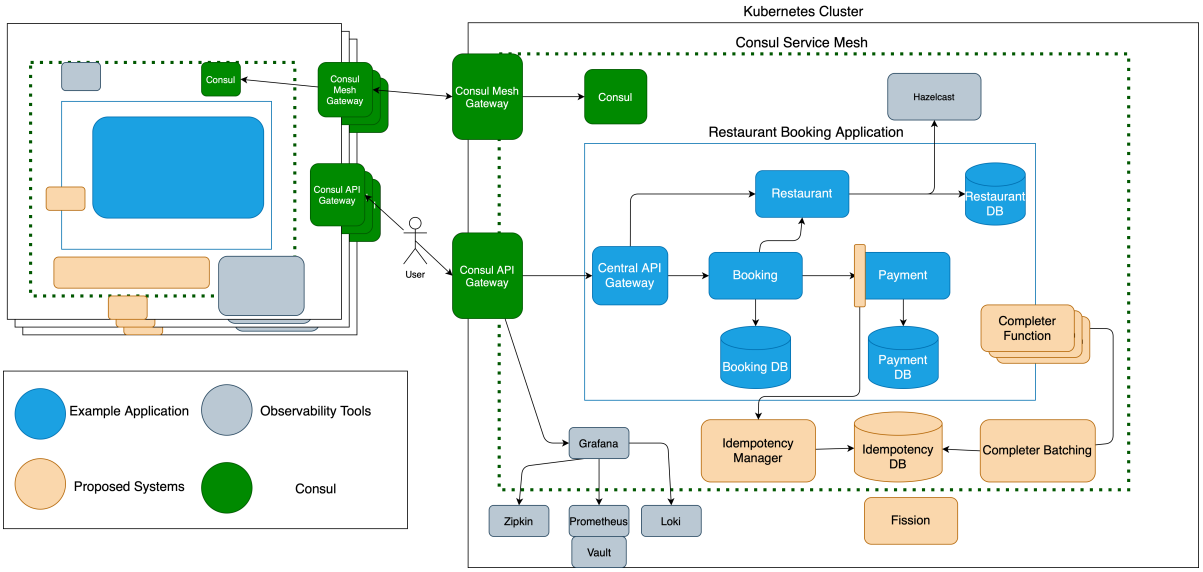


Figure 1: Example microservice fog-native application for restaurant reservations, showing further the underlying middleware services for managing the application within a Kubernetes cluster and connecting it to microservices in other clusters over Consul service mesh.

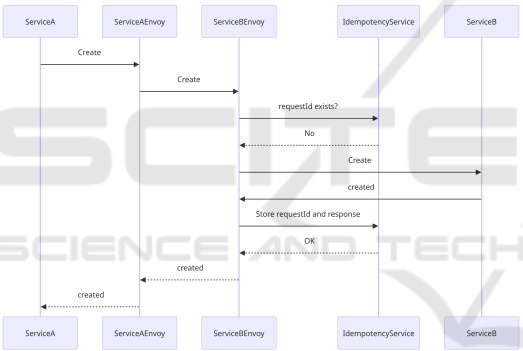


Figure 2: Idempotency request not already processed.

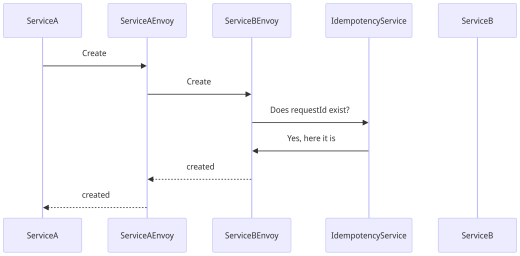


Figure 3: Idempotency request already processed.

idempotency entries. Notably, although idempotency state is maintained within the system, the IM itself is designed to be stateless and horizontally scalable because management of the idempotency state is offloaded to a database system. This allows the IM to be fast and lightweight. One endpoint of the IM is a `GET` that takes as input: an idempotency key, service name and request digest. First, the idempotency key and service name are extracted from the request. The two provide a unique composition that allows for injecting multiple idempotency entries, for a given request from different services. Then, a lookup is performed over the idempotency key and service name, in the database. If an entry is found, the request digest is examined to verify if the received digest matches the stored one. If the two match, the response details are extracted and returned to the filter as described earlier. Otherwise, i.e. if the two digests do not match, the endpoint responds back to the filter with a status code `409`, indicating a conflict of idempotency.

The second endpoint of the IM is a POST that interacts with the idempotency database to inject new entries. The POST endpoint takes as input: the idempotency key, the service name, cluster name, response body, response status code, request digest and completer downstream resource (described in Section 3.1.3), if present. These are used to construct a new idempotency entry in a database-compliant JSON format. The new entry is added along with the current timestamp to allow for cleanup to mitigate idempotency key conflicts. The functional flow for both endpoints is illustrated in the papers Github.

The idempotency information base is implemented over a database system, to offload state synchronisation along with other management operations to the database platform, keeping the IM stateless. This exploits existing capabilities of database systems to synchronize across clusters in near real-time. A PostgreSQL relational database is used here as an ex-

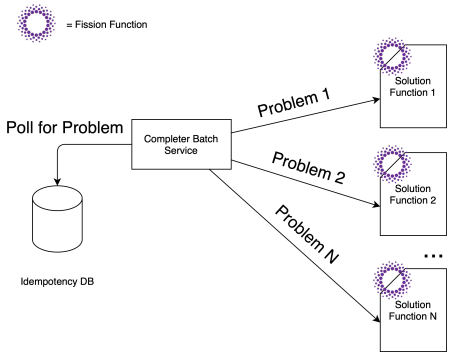
ample, for its scalability, as observed in Active/Active database systems, as well as the benefits of ACID transactions. The database schema is defined in the aforementioned GitHub repository.

3.1.3 Completer System

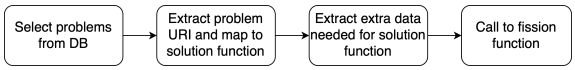
The Completer system illustrated in Figure 4 handles non-transient errors due to microservice defects, that the idempotency system cannot resolve. It provides automatic and systematic completion of requests, without manual intervention or ad-hoc temporary solutions, specifically when an error occurs within a service while processing a request. Here, the IM would have processed and saved the error, and all subsequent automated retries of the request by Envoy will yield the same result. However, with the completer, if a problem and a solution to that problem are known in advance, a successful response can be sent back to the client and the retry can be issued at a later time once the error within the service has been rectified. This is to allow the request to succeed asynchronously. The approach works well for use-cases of aggregated JSON resources, made up of sub resources that were not the main purpose of the request but are necessary to its successful completion. If such sub resources are not required immediately by the user, they can be handled by the completer and have the full resource ready for when needed for example, an error occurs with the payment service, a successful booking can still be made, while the payment issue is resolved later by the Completer.

To achieve the above, the Completer requires services to conform to RFC9457 (Nottingham et al., 2023). This defines a machine-readable type member, used here to commonly identify a specific problem and map directly to a solver FaaS function. Furthermore, to allow a completer function to create a valid solution, additional information may need to be obtained from a downstream service about the upstream request. In order to facilitate this, a new header is defined `completer-downstream-resource`. This header is formed of a list of key-value pairs separated by semi-colons, following the same format as the `x-forwarded-client-cert`, defined by Envoy.

To allow developers of a microservice to define their own solutions without coupling the Completer with the microservice, a FaaS approach is adopted in offering solvers - illustrated in Figure 4a. Here, developers write modular and self-contained functions that solve specific problems, independent from each other, and expose them as a service within the Completer. In order for the relevant FaaS solvers to be called, a central batch microservice is developed within the completer. The batch microservice performs a one-to-one



(a) FaaS Architecture.



(b) Batching Pipeline.

Figure 4: Completer System.

exact matching between an idempotency problem and the FaaS solvers. If a match is found, the solver is called to address the problem. Subsequently, the only point of the completer that needs to be updated when adding a new solver function, is the map of problems to solvers.

Figure 4b illustrates the workflow of the Completer. First, the batch service queries the Idempotency Information Base for entries with incomplete requests. For each entry, the batch service verifies: if the entry has already been processed; and, if the response is of type “ProblemDetails”. If so, the batch looks up a matching solver function for the problem, using the problem URI, and extracts any further data needed from the error and the `completer-downstream-resource` header. The batch then calls out to the fission router using the HTTP trigger associated with the problem. Notably, fission is used here merely as an example for FaaS management platform. However, the batch service is not coupled to fission and can be integrated with any FaaS platform that allows an HTTP endpoint, which accepts POST messages.

4 EVALUATION

This section evaluates the performance of the proposed solution experimentally, using the application of Figure 1. It compares the performance of the FNA deployment over multiple clusters, with and without the proposed idempotency-completer solution. The experiments are deployed over an elementary testbed. in the Network Convergence Laboratory (NCL) of the

University of Essex [3]. The testbed is comprised of two clusters, each including two machines. Standard Consul failover is enabled as a default resiliency measure, both within a cluster and across clusters. The upper bounds of 18 users per second is a limitation of computational power in our testbed and should not be seen as a limitation of the proposed systems. Given an environment with a production-level PostgreSQL database cluster among others, the throughput of the system would be much higher. Nevertheless, the results still show the capability of the system.

The four machines are of similar capacities, and connected by a standard 1Gb/s Ethernet switch. Each cluster is locally orchestrated by its own Kubernetes, with K3s chosen as a lightweight flavour of the orchestrator. Consul Connect cluster peering is used to facilitate cross-cluster communication. The choice of only two clusters is merely for simplicity. The system can technically scale beyond this, exploiting the multi-cluster peering capability of Consul. Each experiment is run with an increasing number of users, to scale up the offered load. Each user generates 1 request per second, having 30 requests over the benchmark period. Request generation is realised using Gatling [4], a widely used load testing tool.

Three Key Performance Indicators were analyzed for the Idempotency and Completer systems: **User Satisfaction**, **Cost Savings from Duplicate Resources** and **Solution Overheads**. The first is measured by the number of lost requests when disabling 'retries', as opposite to enabling safe retries with the idempotency system. The KPI is measured for the Completer by the total number of completed requests as opposite to the ones sent (results in the GitHub repository). **Cost Savings** is measured by the number of prevented non-safe retries that would have caused duplicate booking of user resources. **Solution Overheads** is measured by the observed response time. Each experiment is repeated for 5 times, for each benchmark setting.

4.1 Idempotency System

This section illustrates the benefits of the IM in enabling safe retries, in contrast to nondiscriminatory disabling of all retries, to avoid idempotency storms.

[3]https://www.essex.ac.uk/departments/computer-science-and-electronic-engineering/research/communications-and-networks

[4]http://gatling.io

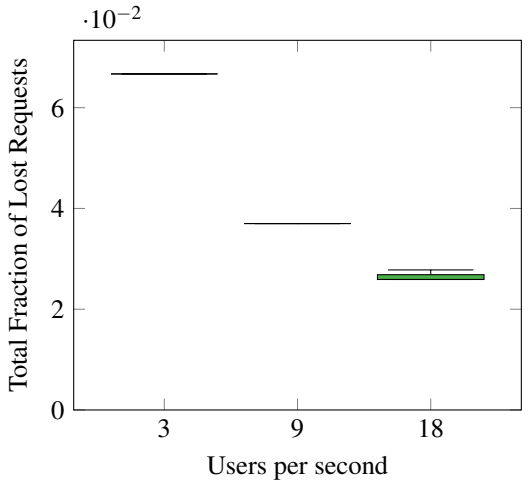


Figure 5: Fraction of lost requests that would have been safely retried if the idempotency system is enabled.

4.1.1 User Satisfaction

User satisfaction is analyzed when one of two instances of a microservice fails and is being evicted/re-added to the load balancer of the mesh. Here, the failure is in the infrastructure not the microservice logic, hence a retry to the other instance is idempotency-safe. Figure 5 shows the fraction of lost requests over the benchmark period, without the IM and when retries are disabled. Notably, when the IM is operating and retries are enabled, no requests are lost due to infrastructure failure. Without retries and the IM, having the outlier detection and circuit breaker patterns of Envoy, the failing instance is evicted after 3 failed requests. Envoy's circuit breaker tries to bring the instance back into the load balancer after a base ejection time (10s in this case), with exponential backoff. This will cause more requests to fail.

Hence, when the number of users is 3 or 9, the fraction of lost requests is constant to 0.067 and 0.037, respectively. However, when the number of users is 18, the loss varies between 0.026 and 0.028. This is because when the number of requests within a time period increases, the service is restored for short periods of time before it's ejected again which results in lower number of lost requests than offered ones.

4.1.2 Cost Savings from Duplicate Resources

This section shows the scale of savings in duplicate resources from non-safe retries, due to application failure. Here, two instances of a microservice are deployed in each cluster, C1 and C2. The two instances of C1 both have a functional error at some point, causing Envoy to retry requests to C1 instances. The instances in C2 are healthy, but Envoy does not redi-

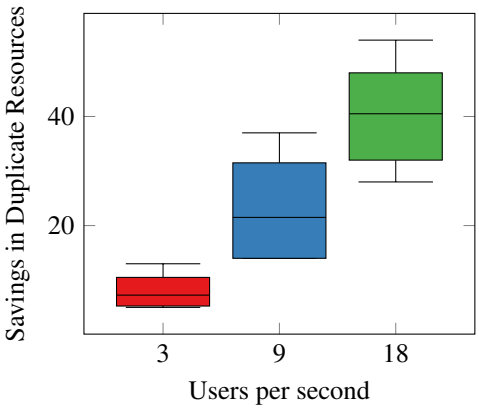


Figure 6: Number of duplicate resource savings by the Idempotency system.

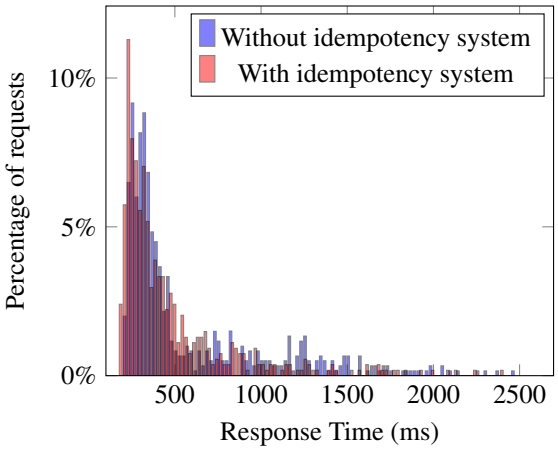


Figure 7: Requests distribution by their response time, with and without enabling the Idempotency system.

rect C1 requests to them. Because, C1 instances are still active as the infrastructure is healthy. The IM is disabled to allow for quantifying duplicate resources due to such non-safe retries, which would otherwise be saved when the IM is enabled. Enovy configurations are described in the GitHub repository, due to space limitation.

Figure 6 shows the savings in end-user duplicate resources over the benchmark period, achieved by the IM when preventing idempotency non-safe retries. In the context of our example FNA, these are duplicate payments charged to clients accounts. They are as well the number of duplicate state that the FNA and infrastructure provides need to resolve manually and maintain 'exactly-once' semantic. The results illustrate an almost linear growth in the number of saved duplicates with the increase in number of requests. The variation seen in the results is due to Envoy's fully jittered retry algorithm[5] that has a certain random delay in retrying selected compounded on multiple levels of the application flow.

4.1.3 Solution Overheads

Figure 7 shows the percentage of requests successfully achieved within a certain response time, when the service mesh is with or without the idempotency system. The results illustrate the overhead of the idempotency system on the end-to-end workflow of a request, including Envoy operations and the FNA task execution. Load is fixed here to 18 users/s. The results show the current implementation adds ≈ 59.626ms per idempotency-safe microservice. This is recognised as a marginal overhead only here, where the application response time is around 580ms. However, for time-critical applications, such as some of 6G scenarios where response time need not to exceed 10s of ms, optimised software engineering is needed to reduce this overhead.

[5]https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/router_filter#x-envoy-max-retries

4.2 Completer System

This section analyzes the Completer overhead compared to the gains in user satisfaction. Results of the latter are provided in the aforementioned GitHub repository.

4.2.1 Solution Overheads

We quantify the overhead of the Completer by the number of requests completed within a time frame. Note that this measure is directly dependent on the CPU and memory specifications of the machine(s) running the Completer. Given our testbed specifications, on average the Completer has been able to process 500 requests in 5s period across all experiments. It is worth noting that Fission (the FaaS platform) introduce a marginal, controllable, overhead (Fission, 2024).

5 CONCLUSION

Resiliency of Fog-native applications, distributed over multiple edge-cloud autonomous systems is a critical challenge because the scale of inter-microservice exchanges elevates the likelihood of failures to unprecedented levels. Service mesh frameworks address many of the failure scenarios, but they fall short in achieving idempotency-safe retries for HTTP POST-based microservices. This work pro-

posed a novel two-part fog-native resiliency solution, providing idempotency-safe retries and automatic completion of requests for HTTP POST microservices. The proposed solution first enables idempotency-safe retries following infrastructure failure, while preventing non-safe retries following application functional failure. The completer part of the solution, overcomes functional failure through automatic application of developer-defined solvers via FaaS. The two parts were implemented as microservice patterns. They have been integrated with Consul service mesh and Fission FaaS platform, as example enablers. The proposed solution has been evaluated experimentally. The results have shown $\approx 0.03 - 0.07$ improvement in request satisfaction due to safe-retries, compared to no-retries. The results have further shown 100% request completion rate, facilitated by the Completer system. This illustrated the benefits of the proposed solution in enabling resilient operation of fog-native applications.

ACKNOWLEDGMENTS

This work has been partially co-funded by the Smart Networks and Services Joint Undertaking (SNS JU) and the UK Research and Innovation (UKRI), under the European Union's Horizon Europe research and innovation programme, in the frame of the NATWORK project (Grant Agreement No 101139285).

REFERENCES

Amiri, Z., Heidari, A., Navimipour, N. J., and Unal, M. (2023). Resilient and dependability management in distributed environments: a systematic and comprehensive literature review. *Cluster Computing*, 26(2):1565–1600. https://doi.org/10.1007/s10586-022-03738-5.

Chandramouli, R. (2022). Implementation of devsecops for a microservices-based application with service mesh. Technical report, NIST.

Consul (2024). Enabling peering control plane traffic. https://developer.hashicorp.com/consul/docs/connect/gateways/mesh-gateway/peering-via-mesh-gateways.

de O. Júnior, R. S., da Silva, R. C. A., Santos, M. S., Albuquerque, D. W., Almeida, H. O., and Santos, D. F. S. (2022). An extensible and secure architecture based on microservices. In *2022 IEEE International Conference on Consumer Electronics (ICCE)*, pages 01–02.

Fission (2024). Define correct resource request/limits. https://fission.io/docs/installation/advanced-setup/#define-correct-resource-requestlimits.

Furusawa, T., Abe, H., Okada, K., and Nakao, A. (2022). Service mesh controller for cooperative load balancing among neighboring edge servers. In *2022 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6.

Gabrielson, J. (2019). Challenges with distributed systems. https://aws.amazon.com/builders-library/challenges-with-distributed-systems/#Distributed_bugs_are_often_latent.

Gattobigio, L., Thielemans, S., Benedetti, P., Reali, G., Braeken, A., and Steenhaut, K. (2022). A multi-cloud service mesh approach applied to internet of things. In *IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6.

Karn, R. R., Das, R., Pant, D. R., Heikkonen, J., and Kanth, R. (2022). Automated testing and resilience of microservice's network-link using istio service mesh. In *2022 31st Conference of Open Innovations Association (FRUCT)*, pages 79–88.

Kubernetes (2024). Multicluster services api. https://multicluster.sigs.k8s.io/concepts/multicluster-services-api.

Leach, B. (2017). Implementing stripe-like idempotency keys in postgres. https://brandur.org/idempotency-keys.

Mendonca, N. C., Aderaldo, C. M., Camara, J., and Garlan, D. (2020). Model-based analysis of microservice resiliency patterns. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 114–124.

Microsoft (2022). Retry storm antipattern. https://learn.microsoft.com/en-us/azure/architecture/antipatterns/retry-storm.

Netflix (2018). How netflix increased developer productivity and defeated the thundering herd with grpc. https://www.cncf.io/case-studies/netflix.

Nottingham, M., Wilde, E., and Dalal, S. (2023). Problem Details for HTTP APIs. RFC 9457. https://www.rfc-editor.org/info/rfc9457.

Prokhorenko, V. and Ali Babar, M. (2020). Architectural resilience in cloud, fog and edge systems: A survey. *IEEE Access*, 8:28078–28095.

Saleh Sedghpour, M. R., Klein, C., and Tordsson, J. (2022). An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 17–27, New York, NY, USA. Association for Computing Machinery.

Services, A. W. (2024). What is a service mesh? https://aws.amazon.com/what-is/service-mesh/#seo-faq-pairs#how-does-a-service-mesh-work.

Shahid, M. A., Islam, N., Alam, M. M., Mazliham, M., and Musa, S. (2021). Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment. *Computer Science Review*, 40:100398. https://www.sciencedirect.com/science/article/pii/S1574013721000381.